QUALITY-ORIENTED SOFTWARE PRODUCT LINE

By
Lei Tan
(MSc)

A thesis submitted in total fulfilment of the requirement for the degree of Master of Philosophy

School of Electrical Engineering & Computer Science
The University of Newcastle
Callaghan 2308, NSW, Australia
May, 2013
Abstract

In software engineering, quality evaluation and achievement are difficult tasks because of the complexity of the software systems. Software reuse is one of the most promoted ways to improve software quality. Software Product Line Engineering (SPLE) is a newly established reuse-based paradigm, which has been well-recognised by the industry. SPLE has been successfully applied by the companies such as HP, Philips, Siemens, TomTom and so on.

Instead of developing individual software products from scratch, SPLE aims to develop a set of similar software systems which share commonalities within a particular application domain. In software product lines (SPLs), reusable assets are developed from the beginning with the view that they will be used in other similar software products. Once they have been successfully developed, the individual product development follows a rigorous customisation process.

Quality-related issues for product lines, on both requirement and architectural levels, are the main focus of my research. To enhance quality-oriented product configuration, we have proposed an approach of measuring the contributions of software features to quality attributes. Features are compared in a pair-wise fashion, and the result is used to generate a ranking list, in which is indicated the relative importance of features to software quality achievement. The ranking list of features is able to greatly help software engineers to understand the factors that impact on final quality, thus assisting product configuration of SPLs. Additionally, the efficiency of feature-based configuration should also be improved, as configuration is normally a time-consuming and error-prone task. To improve the efficiency of configuration, we have taken into account of the dependencies between features,
and adapted some classical algorithms to reduce errors and rollbacks possibly occurring in
the product configuration. We have also considered quality issues in the process of soft-
ware product line architecture development. A quality-oriented architectural framework
has been proposed to specify various views and components composition for improving
the quality awareness at the architectural level.

We believe that software quality should be emphasised and modelled throughout the
whole process of SPL development, rather than been focused on in a particular phase
in the development. We have proposed an aspect-oriented SPL framework, in which
we have introduced aspect-oriented modelling for both feature modelling and reference
architecture design. The proposed framework is expected to model the impact of the non-
functional requirements (NFRs) better, and to deal with software quality from requirement
engineering to architecture design in a systematic way in SPL development.
DECLARATION

The thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. I give consent to the final version of my thesis being made available worldwide when deposited in the University’s Digital Repository**, subject to the provisions of the Copyright Act 1968. **Unless an Embargo has been approved for a determined period.
ACKNOWLEDGMENTS

I would like to thank my supervisors Dr Yuqing Lin and A/Professor Huilin Ye for their support and guidance over the past two years. Their encouragement, constructive criticism, and incredible enthusiasm have provided the basis for these studies.

Thanks must go to my colleagues and friends, Guoheng Zhang and Peng Wang for their advice and experience which make this thesis possible.

I wish to thank my wife Chenchen Jiang, and my kids Jason and Alison, who accompanied and supported me through the whole process.
Table of Contents

1 Introduction 8

1.1 Introduction ................................................................. 8
1.2 Problem Statement .......................................................... 10
1.3 Overview of Research ...................................................... 12
1.4 Contributions ............................................................... 15
1.5 Structure of Thesis ......................................................... 17
1.6 The List of Publications .................................................... 18

2 Background 20

2.1 Software Product Line Engineering ..................................... 20
2.2 Domain Engineering ....................................................... 22
  2.2.1 Variability Management .............................................. 23
  2.2.2 Feature Modelling ................................................... 24
  2.2.3 Reference Architecture ............................................. 27
2.3 Application Engineering .................................................. 29
  2.3.1 Product Configuration .............................................. 29

3 Quality-Oriented Feature Ratings 32

3.1 Introduction ............................................................... 32
3.2 Literature Review .......................................................... 34
  3.2.1 Non-Functional Properties Prediction .......................... 34
  3.2.2 F-SIG ................................................................. 35
# 3.2.3 Quality Prediction for Product Lines ........................................ 36
# 3.2.4 Summary ..................................................................................... 37
# 3.3 AHP-based Approach ................................................................. 38
# 3.4 The Elo Rating System ............................................................... 39
# 3.4.1 Approach Overview ............................................................... 41
# 3.4.2 Case Study .................................................................................. 44
# 3.5 Summary ....................................................................................... 51

4 Efficient Product Configuration ....................................................... 53
4.1 Introduction .................................................................................... 53
4.2 Literature Review .......................................................................... 55
4.2.1 Feature Model Analysis ........................................................... 55
4.2.2 Product Configuration Approaches .......................................... 57
4.2.3 Summary .................................................................................... 59
4.3 Approach Overview ................................................................. 60
4.4 Vertex Cover Problem and Simulated Annealing Algorithm ........... 65
4.5 Experiments and Results ............................................................ 67
4.5.1 Library System Feature Model ............................................... 67
4.5.2 Random Systems ..................................................................... 73
4.6 Summary ....................................................................................... 75
4.6.1 Effectiveness of Our Approach ............................................... 75
4.6.2 Future Work .............................................................................. 75

5 Quality-Oriented SPL Architecture .................................................. 77
5.1 Introduction .................................................................................... 77
5.2 Literature Review .......................................................................... 78
5.2.1 FORM ......................................................................................... 78
5.2.2 PuLSE ......................................................................................... 79
5.2.3 KobrA ......................................................................................... 80
5.2.4 Model-driven Approach ........................................... 81
5.2.5 Pattern-based Approach ........................................... 81
5.2.6 Summary ......................................................... 82
5.3 QADA ............................................................... 83
5.4 Enhancements of QADA ............................................. 86
   5.4.1 Quality View ................................................ 87
   5.4.2 Tradeoff View ................................................ 89
5.5 Case Study ......................................................... 92
5.6 Aspect-Oriented Architecture Design Approaches ........... 96
   5.6.1 Aspect-Oriented Architecture Approaches for Non-SPL Systems ...... 96
   5.6.2 Aspect-Oriented SPLA Approaches .......................... 99
5.7 Proposed Framework .............................................. 103
   5.7.1 Aspect-Oriented Feature Model ............................. 104
   5.7.2 Aspect-Oriented Reference Architecture ................. 107
5.8 Summary .......................................................... 110

6 Conclusion and Future Work ....................................... 112
6.1 Conclusion ....................................................... 112
6.2 Future Work ...................................................... 114
### List of Figures

2.1 The Framework of SPLE .............................................. 21  
2.2 An Example of a Feature Model .................................. 25  
2.3 Examples of Dead Features ....................................... 26  
3.1 Rating update for “Ease of Use” ................................. 49  
3.2 Actual Scores for Each Feature of “Ease of Use” ............ 50  
4.1 A Variation Point (V_P) and its Variants ....................... 62  
4.2 The Dependencies among Variants ............................... 62  
4.3 Transform a Feature Model into a Directed Graph ............ 65  
4.4 A Feature Model of Library Systems ............................ 68  
5.1 Our Approach contributes to QADA ............................... 87  
5.2 Developing Quality View ......................................... 88  
5.3 Developing Tradeoff View ....................................... 89  
5.4 The Structure of PACS ............................................ 92  
5.5 The Partial Reference Architecture of PACS .................. 93  
5.6 Security and Performance Tradeoff .............................. 95
List of Tables

3.1 Quality Attributes and Contributing Features ...................... 44
3.2 Relative Importance Matrix given by the First Domain Expert .......... 46
3.3 Fixed Matrix given by the First Domain Expert ........................ 46
3.4 Relative Importance Matrix given by the Second Domain Expert ........... 47
3.5 Feature Comparisons given by the First Domain Expert ................. 47
3.6 Updated Rating based on the Opinion of the First Domain Expert .......... 48

4.1 The Dependency Relationships among Variants in a Feature Model .......... 69
4.2 Max Configuration Coverage of each VP ............................... 70
4.3 A Sequence of VPs which covers Feature Model ......................... 70
4.4 A Sequence of VPs after a Decision made at VP8 ........................ 71
4.5 Depth-First Configuration Group(left) and Optimisation Configuration Group(right) 72
4.6 Group Comparison of Average Figures .................................. 73
4.7 Experiment Result of Random Systems ................................. 74

5.1 Quality Attributes Relationships ..................................... 90
5.2 Architecture Options by Tradeoff ..................................... 91
5.3 Quality Attributes and Related Components (connectors, ports) ............ 93
Chapter 1

Introduction

1.1 Introduction

The nature and complexity of software have changed significantly in the last 30-40 years, and software engineering techniques need to adapt to keep up. Software Engineering (SE) includes problem-solving techniques for software systems design, development, or maintenance. SE starts with obtaining a problem statement from clients or stakeholders, and ends with acceptable software being delivered. SE aims to manage this development process in a systematic and disciplined way.

Software quality is always considered as a critical factor of software development. A successful software product has not only to satisfy a functional requirement, but must also be reliable, robust and maintainable, i.e. with high quality. Current software development is commonly a multi-iteration process, in which achieving high quality in the early iterations will considerably reduce the effort of modification and re-development in the later phases. This is one of the main reasons why software quality is so emphasised in the early stage of software development. However, the quality modelling in the early stage of software development is difficult, as available information is inadequate at this stage.
Furthermore, software assets developed in the latter stages, e.g. architecture design, also have impact on the final quality.

To preserve software quality and to improve the efficiency of software development, reusing software and its components has become a practical solution as many software organisations are specialised in particular application domains, and thus have adequate experience of similar systems as well as access to the previously developed products. Software reuse should reduce redundant development efforts. At the same time, it should improve the overall quality of software by reusing designs and components from previous successful implementations. The recognised benefits of software reuse include reduced cost and time, improved productivity and software quality, improved maintainability of applications, and wider shared knowledge and learning (Rotheberger and Nazareth, 2002).

**Software Product Line Engineering (SPLE)** is a software reuse paradigm that focuses on systematic software reuse (Bosch, 2000). SPLE deals with a set of similar software systems as a software family that contains a common set of software elements satisfying various requirements within a particular application domain. SPLE contains two main processes: domain engineering and application engineering. The activities which create a software product line (SPL) and related assets belong to domain engineering. The activities which produce individual software products from an SPL are included in application engineering. SPLE enables the development of reusable software assets which are configured and modified for producing software member products.

In SPLE, all the reusable assets are developed in domain engineering, so it serves as a basis, which is expected to facilitate product derivation in subsequent application engineering. Tasks in domain engineering are critical to SPL development, and issues such as variability modelling, quality management, and architecture development, need to be well managed.
1.2 Problem Statement

There are two essential reusable assets developed in the domain engineering phase of S-PLE: “feature model” and “reference architecture”. Feature model is a key artefact of S-PLE, which indicates the commonalities and differences among the member products on the requirement level. A feature model represents all the possible member products of an SPL, in terms of features and the relationships among features (Kang et al., 1990). A feature model is mainly used to represent available information of an SPL on the requirements level. It is also used to configure member systems by selecting desired features to reflect particular requirements of a member product. A software reference architecture provides a common architectural structure, related components and their interactions with an application domain of an SPL (Eixelsberger et al., 1998). A reference architecture provides architectural templates for an SPL and it will be configured and refined to produce the concrete architectures for member products. S-PLE preserves systematic mappings from requirements to architectures, to ensure that requirements and expectations will be achieved by the final products.

As mentioned previously, the achievement of software quality is always a critical concern in software development. Quality factors are difficult to assess, and often evaluated after the system has been developed, because it is a challenge to assess quality factors of the final product at an early state of the system development, where the software assets are not fully developed. For example, maintainability is an important property of the software. To build a maintainable system, efforts need to be made in different development stages. In the design stage, architects should plan the software structure to be easily altered. In the implementation stage, engineers should create code in an easily understandable way. It is hard to evaluate the maintainability of a system at an early stage or any particular
phase of software development. On the other hand, evaluating quality factors after the
system is fully developed is also risky. If related quality criteria does not meet the ex-
pectation, it could lead to major modifications to the system, which are expensive and
highly undesirable towards the end of system development. So it is important to take into
account quality factors, during the development process and consider them in terms of
their relationships with various software assets from different development stages.

In the context of SPLE, quality issues are even harder to deal with, because of the
existing variability between member products. Quality factors are not treated and handled
properly in current SPLE (Zhang et al., 2010). In the event that expected quality and
NFRs are not achieved by the final products, it is likely that a re-configuration process has
to be carried out, which is less desirable. Similar to the situation in non-SPL systems, to
better assess the quality of the product, it is necessary to specify the relationships between
quality attributes and software assets on both requirement and architectural levels, i.e.
feature modelling and reference architecture development, during SPL establishment and
product derivation.

As feature model-based product configuration is the very first step in preventing invalid
products, which will affect the further transformation on the architectural level, a quality
estimation and prediction of the final product should be conducted to a certain degree at
this stage. To have accurate quality estimation, it is necessary to measure the relationships
between functional features and quality attributes, e.g. the contributions of an individual
feature to the corresponding quality attributes, or the aggregation of the impact of a set of
features in a configured product. To minimise the possibility of invalid configuration, the
impact of various features on each quality attribute should be identified before actually
configuring a product.

Except quality concerns, how to configure member products efficiently is another issue
of SPLE. Feature model-based configuration is a time-consuming and error-prone task, due to the complexity caused by the large number of features and relationships between features. Conventional product configuration adopts the depth-first traversal approach, i.e. configuration starts from any hierarchy of the feature model and makes a decision on each feature until reaching the end of this hierarchy. Then it will move to the next hierarchy to repeat this process until all the features have been visited and determined. Imagine in a real-world feature model, such as the Linux Kernel feature model, with over 6000 features and 8000 constraints among these features (Hubaux et al., 2011), it would be extremely difficult to configure in such a conventional way. The constraints between features also restrict the selections during configuration; however, this information is not considered in conventional product configuration.

On the architectural level, when conducting configuration on the reference architecture to produce architectures of member systems, the quality-related derivations are particularly important. It is crucial for the reference architecture to support requirements traceability, i.e. providing explicit information on how the quality requirements are realised in the software architecture. This would give the software engineers a degree of confidence in proceeding, knowing that the quality requirements had been considered and modelled in the architecture. Furthermore, since member products have different desired qualities, so the reference architecture should provide architectural options for satisfying quality attributes at different levels, and highlight the restrictions and relationships among these options.

1.3 Overview of Research

We have concentrated our effort on quality-related issues in current SPLE, from several points of views. Firstly, we expect to model NFRs and quality attributes in the stage
of domain engineering, and to identify the relationships (contributions) between quality factors and SPL assets, e.g. quality attributes and features, quality attributes and components. The relationships between software elements and related quality attributes will be identified to preserve the traceability that explains how NFRs are achieved in final products.

Secondly, we are also interested in improving the efficiency of product configuration. We intend to simplify product derivation and also minimise errors and invalid configurations. So to improve conventional product development, configuration should be conducted by taking into account both feature constraints and their contributions, to achieve quality requirements.

Moreover, we are going to investigate how to improve systematic mappings and transformations from requirements (features) to architectures (components). Appropriate mechanisms are needed to represent this information explicitly in order to achieve efficient product derivation. Variability management is what needs to be enhanced to represent the unique requirements of member systems clearly.

In this research, we have addressed some of these tasks. The research focuses on quality-related issues, and on improving the efficiency of product configuration. For quality-oriented product configuration, a ranking of features based on their contributions, would be helpful to provide a general idea of how important each is in the achievement of a particular quality attribute. Suppose that there are various ranking lists of features available for different quality attributes. It would be advantageous for users to understand how to achieve overall software quality, and how to treat tradeoff issues in quality attributes. We have adapted the Elo rating system (Elo, 1978) to conduct pair-wise comparisons to evaluate the relative influence of two features on a quality attribute. By integrating ranking lists from different sources, it is possible to provide a direct picture of how much each
feature affects quality attributes. When users configure a product, the comparison results guide users to consider quality achievement rigorously. These results provide users with a general idea of how to treat tradeoff issues to ensure that quality attributes are achieved by appropriate feature selections.

To improve the efficiency of product configuration on the feature model, we have considered feature relationships and simplified configuration steps. The key idea in our approach is to identify a small set of key features. The selections of these key features will determine all the other features, based on their interrelationships. Thus, we have reduced the number of variation points to visit during the configuration process. As a result, the decision steps and rollbacks are significantly reduced. To identify this set of key features, we transform the feature dependencies as a directed graph and then model the problem as a minimum vertex cover problem. Then we apply the HSAGA algorithm (Tang et al., 2008) in finding the minimal coverage of the feature model. As shown by the results of experiments, our approach saves configuration efforts, and the improvement in configuration efficiency is clear.

To achieve the ultimate goal of systematic software development, requirements need to be properly mapped and transformed onto the architectural level. A quality-oriented reference architecture development is expected to address all the corresponding quality issues on the architectural level. To improve such a paradigm, we have proposed an approach that introduces some extensions to an existing SPL architecture method, known as “Quality-driven Architecture Design and quality Analysis” (QADA) (Matinlassi et al., 2002). The proposed extra views are the “quality view” and the “tradeoff view”. The quality view is designed to provide requirement traceability in the reference architecture. At the same time, it describes the impact of quality attributes on the reference architecture. The quality view contains many subsystems. These subsystems contain the information describing
how the components interact with each other. For each quality attribute, communication-
s between the related components are identified and included in the subsystems. Since the components are configurable, the quality view also contains options for satisfying the quality requirements at different levels. The purpose of the tradeoff view is to deal with trade-offs in quality attributes, to help product configuration. One tradeoff decision could cause big changes in various parts of the system architecture, thus each one has to be managed properly. The tradeoff view includes the information of quality-related decisions and functional selections. The tradeoff view can be traced back into the quality view and used for member product configuration.

1.4 Contributions

This research has resulted in contributions to requirement and architectural levels of SPLE development. In this research, quality issues have been considered as the main drivers, and have an impact throughout the development process.

- **Enhancement of feature-based quality assessment**

  To enhance quality assessment of features, we have proposed an approach by adapting the Elo rating algorithm which was originally developed for chess players’ ratings. Compared to other quality-assessment approaches, the Elo-based approach would be able to consolidate information from different sources, and it is also able to tolerate small variation of opinions from domain experts. We have compared our approach with others and believe that it provides a unified contribution towards feature-based quality assessment.

- **Improvement of valid software configuration**

  The complexity of feature-based configuration has been recognised because of the
sizes of feature models and the relationships between features. So the key to improving configuration efficiency is managing feature relationships to prevent potential errors. Such an approach is developed and evaluated; the theory and implementation algorithm are explained and used to produce a configuration framework.

To improve the efficiency of identifying key feature set, we adapted the HSAGA algorithm in case of large feature models. We have applied the algorithm to randomly-generated feature models to simulate real-world cases to evaluate our approach. According to the results of experiments conducted, this approach saves 21.7 per cent of configuration effort compared to conventional configuration process, and configuration conflicts are managed properly. So the overall quality of product configuration is improved. In investigating related paradigms, we have not found any approach with similar ideas, so we claim that our approach provides an efficient and unique method of configuration management in the context of SPLE.

- **Quality specification for architecture development**

  As a systematic software development paradigm, quality issues on the requirement level need to be mapped and inherited on the architectural level. A reference architecture development approach is proposed, based on an existing quality-driven SPLA approach, in order to provide more specific information of realisation of quality attributes and NFRs. The approach provides two extra views: one describes the relationships between quality attributes and components; the other one deals with complex quality trade-offs on the architectural level.

  This approach is closely linked to the research work at the requirement level and its contribution is considered as an enhancement of quality-oriented SPLA development.

- **Aspect-oriented SPL framework**
We have also proposed a framework which is considered as future work, to model quality and variability in SPLs. The idea of the framework is to adapt aspect-oriented techniques to provide an advanced structure to address quality attributes, and to represent variability more efficiently. More information of this framework is included in Chapter 5.

At this stage, the framework still lacks some details. However, we have done some investigations of related works of existing aspect-oriented approaches. We believe that our method contains a unique idea in terms of a comprehensive framework by introducing aspects to multiple layers of software development. Our framework is able to modularise the impact of quality attributes on multiple features and feature groups, and to reflect this information at the architectural level to preserve SPL traceability in a well-defined fashion.

1.5 Structure of Thesis

The remainder of the thesis is organised as follows:

- **Chapter Two: Background.** This chapter briefly introduces the background of SPLE and the two main processes of SPLE. The critical activities of SPLE are included and discussed to provide a high level understanding of SPLE.

- **Chapter Three: An approach to quality-oriented feature ratings.** This chapter presents our approach to quality-oriented feature ranking, based on its contribution to related quality attributes. The importance of quality-oriented feature rating is discussed and existing issues of related work are addressed. A case study is also included to illustrate the improvements achieved by using our approach.

- **Chapter Four: An approach to improving the efficiency of product con-
configuration. This chapter introduces an approach developed in order to improve the efficiency of product configuration in SPLE. Feature-based product configuration has to be efficient, as a large number of features and complex relationships occur in the practical configuration process. This unique approach takes inter-relationships between features into account in the configuration process, to avoid errors and rollbacks. Moreover, this approach preserves the overall quality of the configuration by preventing potentially invalid feature selections.

• Chapter Five: An approach to quality-oriented SPL architecture development. This chapter presents a quality-oriented architecture development approach to enhancing the systematic architecture development of SPLE. Towards this end, the quality-oriented feature modelling needs to be well mapped and transformed to the architectural level, to guarantee quality achievements in derived products. The quality attribute trade-off issues are also considered, and improved by investigating the relationships between software elements and quality attributes in the particular tradeoff view. A proposed framework of aspect-oriented reference architecture development is also included to describe our idea of how to manage quality issues in SPLE more efficiently.

• Chapter Six: Conclusion and future work. This chapter concludes the thesis by describing the contribution of this research to SPLE. Future work is also discussed and some research questions are listed.

1.6 The List of Publications

1. L. Tan, Y. Q. Lin and H. L. Ye, Modeling Quality Attributes in Software Product Line Architecture, in the proceedings of 2012 Spring World Congress on Engineering
and Technology (SCET2012), Xi’an, China.


Chapter 2

Background

2.1 Software Product Line Engineering

A software product family consists of similar software products. Member systems in a software product family share commonality, meanwhile, they contain some particular features that make them different from other members. For example, banking systems used by different banks may have different interfaces, and interest calculations, but they still provide similar services, such as online banking, balance checking, or cash withdrawal.

Software product line engineering (SPLE) is a software reuse-based approach to developing a software product family and member products. The concept of a software product line (SPL) was first raised by Parnas (1976), and has been further developed by Kang et al. (1998).

Different from ad-hoc reuse, SPLE aims at promoting systematic software reuse (Bosch, 2000). Systematic software reuse refers to the paradigm in which a set of related systems share design decisions in a focused application area, and rely on a repeatable process with the reuse of high-level software artefacts (Frakes and Isoda, 1994). All artefacts developed from an SPL will be reused to produce member products based on users’ requirements.
for different members. So an SPL contains various reusable assets and artefacts, such as system requirements, source code, architecture and documentation. When developing a member product of the family, these reusable artefacts will be refined, modified and configured. SPLE saves the effort of developing every single software system separately. It provides systematic mappings between requirements and reusable artefacts, and preserves the traceability to ensure that requirements will be realised by final products. Once the reusable artefacts are modified and successfully implemented, the individual member products can be derived through a very organised customisation according to the corresponding requirements for the individual member systems.

![Figure 2.1: The Framework of SPLE](image)

Fig. 2.1 demonstrates the main framework of SPL development. SPLE consists of two main processes: Domain Engineering and Application Engineering. There are multiple activities included in domain engineering, such as, domain requirement engineering, domain design, realisation and testing. Overall, these activities will create reusable artefacts for the product family. Moreover, variability of a software product family also needs to be addressed in domain engineering. For example, particular requirements for some member products need to be included in the general requirement document produced during
requirement engineering. The variability of system requirements will be transformed onto the architectural level to reflect the impact of the requirements on the interaction among software components. All the information produced and captured in domain engineering provides a solid picture of a software product family.

The activities which produce individual software products from SPLs belong to application engineering. Application engineering is basically a refinement process applied to the reusable artefacts, according to the information collected for the member product. For example, the particular requirements of each individual system need to be understood, and the general requirement document from domain engineering needs to be modified, to capture this information. Also, the design document in domain engineering needs to be modified accordingly to produce particular design documents and member product architectures. All the refinements are based on the variability addressed, and related changes documented, in domain engineering.

2.2 Domain Engineering

As domain engineering produces the overall scope of SPLs, it is a critical foundation that affects the success of the derivation of software products. In domain engineering, several issues need to be handled carefully because of the complexity of SPLs. As one of the most fundamental tasks, variability management aims to explore and specify the differences between member products from the requirement level and the architectural level. Feature modelling manages both functional and non-functional features on the requirement level. Reference architecture development establishes a common structure for the product families by considering component composition and variability representation on the architectural level.
2.2.1 Variability Management

Variability management in domain engineering involves managing and treating variability between individual software products designed by product line engineering (Krueger, 2002). In SPLs, variability provides the flexibility for customisation and derivation of software products. Variability management maintains the purposes of clearly representing the variabilities in artefacts, managing the dependency relationships between variabilities, and deriving variabilities between software products (Schmid and John, 2004). In domain engineering, domain experts introduce variability mechanisms into software artefacts for the realisation of variations in products. Then application engineers will make design decisions concerning these artefacts, to achieve the required functionalities and qualities for their particular software products.

Variability management also introduces another level of complexity to SPLE, because of the interrelationships between software artefacts. When composing artefacts to produce member products, the relationships between artefacts should be considered. However, the specification of these interrelationships is not straightforward. Moreover, variable options affect one or more quality aspects of each system, and this impact may not be realised until the testing phase (Sinnema et al., 2006). When viewing a large product scope with thousands of artefacts or elements and the potential dependencies among them, it is extremely difficult to clarify which artefact has an impact on which quality attributes. These complicated situations make member product derivation in application engineering time-consuming and error-prone. So powerful approaches and tools are required to support variability management in SPLE.

Towards this end, variability management techniques and tools, such as FODA (Kang et al., 1990), COVAMOF (Sinnema et al., 2004), Kolish (Asikainen et al., 2004) and OVM (Pohl et al., 2005), have been developed and applied to deal with variability issues. These
techniques seek to represent variabilities and manage them in an enhanced and systematic way. More details and discussions of variability management techniques are available in some recent surveys (Chen et al., 2009; Sinnema and Deelstra, 2007; Svahnberg et al., 2005). Chen et al. (2009) conclude that the status of variability management evaluation is poor, as most of the proposed approaches do not have sufficient and rigorous supporting evidence. This is what needs to be improved in variability management.

2.2.2 Feature Modelling

Feature modelling is one of the most important assets of an SPL. It has been widely used to represent all design considerations or factors of a product family. In a feature model, features are prominent and distinctive system requirements or characteristics in an SPL (Lee et al., 2002). Feature modelling was first proposed by Kang et al. (1990) in order to represent features, their relationships, and the constraints of feature selections. A feature model is used to configure member products, and also to develop software architectures.

A member product in an SPL is defined by a unique, valid combination of selected features. A feature model is usually represented as a tree in which the variabilities of features are represented as variation points. In a feature model, a variation point (VP) (Pohl et al., 2005) consists of a parent feature, a group of child features, called variants, and multiplicity to specify the minimum and maximum numbers of variants that can be selected from the variation point when configuring a product. The selection of variants at a variation point is not only constrained by the multiplicity but also by the dependencies between the variants at this variation point and the variants at other variation points.

There are two main types of feature relationships: hierarchical relationships and cross-tree relationships. A hierarchical relationship refers to the relationships between a parent feature and its children features. A cross-tree relationship (Benavides et al., 2010) exists
between non-parental features found in different branches of a feature tree. Hierarchical relationships include mandatory, optional, alternative, and or. The most common cross-tree relationships are requires and excludes. For example, requires is defined by Kang et al. (1990) as: if a feature requires, or uses, another feature to fulfill its task, there is a requires relationship between the two features. Fig. 2.2 is an example of a feature model of a mobile phone product family. In this feature model, Screen is a mandatory feature that must be included for all mobile phones. GPS could be an option for some mobile phones, but it is not necessary for all mobile phones. For cross-tree relationships, Camera “requires” HD means that, if you want to select the Camera function in a mobile phone, HD must also be included. Conversely, “excludes” between GPS and Basic means that, if GPS is used in a mobile phone model, then you cannot have the Basic screen, because of the constraint. However, some feature relationships are complex and hard to measure quantitatively. How to incorporate those complex relationships into feature configuration is a problem.

Multiplicity at a variation point is another important factor that affects feature selections. Feature models have been developed as so-called “cardinality-based” feature models, and new relationships are introduced. For example, a feature cardinality denoted [1..n] means that at least 1, but up to n, features can be included. So a mandatory rela-
tionship can be refined and denoted as [1..1], and an *optional* relationship can be denoted as [0..1]. Potential errors or conflicts could be reduced, by considering the relationships and multiplicities between features, before deriving products.

The quality of the feature model directly affects the quality of the final software products, so the feature model needs to be treated carefully. Improper relationship representations will cause problems in feature modelling, during later development stages. For example, Dead Features commonly exist in a feature model and are not easily detected. A dead feature is a feature which cannot appear in any product of the product line. Dead features are caused by the incorrect use of cross-tree constraints. Fig. 2.3 represents several examples of dead features, denoted in grey in the figure. For the first feature model, mandatory feature B “excludes” alternative feature D, so feature D can not be included in any product derived from this feature model. Similarly, dead features exist in the other two feature models because of improper cross-tree constraints. More analysis of common anomalies in feature models are available in Benavides et al. (2010). Such incorrect constraints should be verified and fixed in order to provide “error-free” feature models as the basis for product development.

![Figure 2.3: Examples of Dead Features](image)

Moreover, current feature modelling includes basically a mix of all the information in an SPL, e.g. functionalities, partial non-functional factors, resources needed, and external responsibilities. Unnecessary information makes feature modelling messy and difficult to understand. It is not clearly specified what should be included in current feature
modelling to facilitate future product derivation. So when modelling features, attention should be focused on how to construct a friendly feature model to improve member product development.

### 2.2.3 Reference Architecture

Software architecture is the structure of a software system. It describes the software elements, their characteristics and also how the software elements interact with each other (Bass et al., 2003). A qualified software architecture becomes a key factor in successful software development, by providing a blueprint for system construction and composition.

For SPLs, systematic mapping between feature modelling and software architecture enhances the requirement traceability of SPLs. To develop architectures effectively for member products, it is necessary to build a reliable common architecture as the basis, which can be further refined to produce the architecture of each member product.

Software Product Line Architecture (SPLA) development is different from the single software system architectural design. More specifically, the member products have different functions and different expected qualities. Very often, from the users’ point of view, these differences are marginal; however, they will result in quite distinct software architectures for member products. So in an SPL, it is common to design a reference architecture with configurable components. A reference architecture (Eixelsberger et al., 1998) is a software architecture that provides common structures, components and their relationships to existing systems in a particular domain or an SPL. The reference architecture and components will be further refined when developing member products’ architectures. Thus, the reference architecture should be flexible, allowing modification to derive concrete architectures for member products.

In software development, non-functional requirements (NFRs) and quality attributes
are the properties of software products. They are concrete requirements that have to be achieved. NFRs are bridges to connect business goals and software architectures (Kazman et al., 2003). For SPLA development, quality issues should be considered more carefully as the quality requirements are various for member systems. So the impact between quality attributes and architectures is harder to manage. When configuring the reference architecture to produce architectures of member systems, the quality-related refinements are crucial to achieve the individual system requirements. It is important for the reference architecture to support requirements traceability, i.e. providing explicit information on how the quality requirements are mapped onto the architecture. This would give the software engineers a certain degree of confidence to proceed, in knowing that the quality requirements have been considered and modelled in the architecture. Furthermore, since member products have different desired qualities, so the reference architecture has to provide various options for the development of member product architectures. Thus, it is useful to evaluate the comparisons between different options and the impact of the options on the rest of the system, e.g. quality tradeoffs. In other words, the reference architecture should provide information such as the architectural options for satisfying quality attributes at different levels, and the relationships among these options.

For SPLA development, there are many approaches and methods that have been proposed. Some approaches focus on member product derivation and architecture configuration; some intend to establish the reference architecture and quality analysis; and also some approaches provide a comprehensive frameworks which covers both domain and application engineering. The well-known SPLA approaches include FORM (Kang et al., 1998), FAST (Weiss et al., 1999), PuLSE (Bayer et al., 1999), QADA (Matinlassi et al., 2002), and AOGA (Kulesza et al., 2004). The details of these approaches will be provided later in the literature review of Chapter 5.
2.3 Application Engineering

Application engineering is the process of producing member products by binding reusable assets according to different system requirements. Application engineering contains several sub-processes and it shows how the existing variabilities are managed to support systematic reuse. As shown in Fig. 2.1, the sub-processes included are application requirements engineering, application realisation, application design and application testing. The goal of application requirements engineering is to elicit the requirements for a member product and assign the requirements to corresponding reusable artefacts. The selected reusable artefacts form part of the requirement specification of the product. The task of application realisation is to configure the member products by using reusable artefacts. For example, a feature model will be configured in the application requirement engineering process, to decide what features should be selected in the products. Then the reference architecture will be modified accordingly in the application design to derive the architectures of member products. Architectures for member products will be derived by integrating selected features into components and subsystems. The final subprocess is application testing, which validates member products against their requirements.

Among these processes, product configuration is a major activity which selects features from the feature model to develop member products. It also serves as the basis for architecture development by mapping features onto components and related interfaces. How to enhance product configuration is a vital issue for our research interests, so we will introduce more details next.

2.3.1 Product Configuration

During the process of product configuration, application engineers specify member products by selecting the desired features. The selection has to follow two rules: 1) it has to
satisfy customers’ requirements; 2) it must obey interrelationships between features. Normally, the application engineers obtain a basic understanding of the requirements of the member products. And the configuration is conducted from the root of a feature model; the application engineers will make a decision on each feature to assess if it is appropriate to include this particular feature into the final implementation of the member product. The selection has to follow the two rules mentioned before. Once we have selected the features to be included in the member product, these features will further suggest other reusable assets for the implementation, such as software components, test cases etc.

Product configuration is a time-consuming and error-prone task, involving manual configuration of products from the feature model. Errors in the configuration process are hard to detect and it is extremely difficult to fix these errors in large and real-world feature models.

Consider a large scale feature model, such as the Linux kernel feature model (Hubaux et al., 2011), which contains over 6000 features. Product configuration would be very difficult. Furthermore, it seems very hard to prevent potential rollbacks and errors in the configuration process if feature relationships are not clearly identified. Once rollback occurs because of invalid selections, it will be time-consuming to trace back to the source of the conflict to correct the mistake. Another problem that needs to be considered is the multi-view issue in product configuration. For example, if different application engineers or customers configure the member products, the final results would vary, even with the same requirements. All of these issues need to be addressed in order to improve product configuration in SPLE.

Because of the complexity of feature-based configuration, researchers have been focusing in different directions to enhance the efficiency of configuration. For example, Staged Configuration (Czarnecki et al., 2005), MUSCLE (White et al., 2009), Priority-based Con-
configuration (Bagheri et al., 2010), and Usage-driven Configuration (Lee and Kang, 2010). The details of these approaches are available in the literature review of Chapter 4.
Chapter 3

Quality-Oriented Feature Ratings

3.1 Introduction

In a feature-based configuration, functional features are relatively well-handled as they are obvious characteristics or functional requirements of product families. On the other hand, non-functional requirements (NFRs) are also very important considerations in product configuration. However, the quality of a final product is always evaluated after the software product is actually configured. If a final product does not meet the quality requirements of customers, a re-configuration process will most likely have to be carried out, which is less desirable. The quality of a target product should be predicted and modelled as early as possible. As feature-based product configuration is the very first step towards preventing invalid products, a quality estimation and prediction of the final product could be conducted at this stage. To estimate the quality of a product, we need to understand software quality of the product family and how individual features contribute to satisfying related quality attributes.

To have accurate quality estimation, it is necessary to understand the relationships between functional features and quality attributes, e.g. the contributions of features to
corresponding quality attributes. To minimise the possibility of an unsatisfactory product configuration, the impacts of features on quality attributes should be identified in the domain engineering process. A feature’s ranking, based on its ability to satisfy quality attributes, would be helpful in providing a general idea of how important the feature is in achieving quality expectations. When there are multiple ranking lists of features for different quality attributes available, it would be beneficial for users to understand how to achieve software quality overall and how to treat quality attributes tradeoff issues.

The key issue in modelling quality attributes in a feature model is to model the relationships between features and quality attributes. A set of approaches has been proposed to measure the interdependencies between features and quality attributes precisely (Etxeberria and Sagardui, 2008; Siegmund et al., 2011; Sincero et al., 2010). The drawback of these approaches is that the real products have to be generated before the evaluation, which is not feasible in practice.

Some other approaches use domain experts’ judgments to predict the relationships between features and quality attributes. Features’ contributions are normally assigned directly by domain experts, with qualitative values. For example, Zhang et al. (2003) proposed an approach of modelling the impact of system variants on quality attributes based on Bayesians Belief Network (BBN), using a BBN model to predict and assess the quality attributes of a configured product. Zhang et al. (2010) have proposed an AHP-based (Saaty, 1980) approach, following the same route, to evaluate features’ contributions, and also to reduce the effort required by domain experts. However, there are still some issues in the AHP-based approach. To deal with these issues, we have proposed an improved approach to enhance the quality-oriented product configuration.
3.2 Literature Review

From the literature describing quality-oriented evaluation, we have investigated and reviewed some related approaches, which focus mainly on quality predictions of the final products, in the context of SPL.

3.2.1 Non-Functional Properties Prediction

Sieg mund et al. (2011) proposed an approach to predicting features’ non-functional properties in an SPL. As features’ non-functional properties are predicted, the non-functional properties of an SPL’s products could be approximated. However, even an SPL with less than a hundred features could produce millions of products, so it is an enormous job to predict the non-functional properties of each product. To improve the efficiency of non-functional properties prediction, this approach investigates alternatives to approximating non-functional properties, based on features’ interrelationships, rather than generating and measuring all products.

To predict the non-functional properties of a product without generating it, each feature’s non-functional properties have to be measured. In this approach, the authors compute a small but suitable set of products, according to the relationships between features. Then each selected feature’s non-functional property values will be approximated by compiling and measuring these products. In other words, the impact on quality of a single feature’s selection will be evaluated. By aggregating the approximations to the desired features, the final product’s non-functional properties will be predicted.

The method of approximating a feature’s non-functional properties is to compare two products and explore the difference in the presence or absence of this feature. The difference between two products will be assigned as the approximation of this feature’s non-functional properties in the form of interval or ratio scales. The process of the ap-
proximation contains several sub-processes. First, a small set of products is generated and measured in an automated process. Then the ratio of difference between two products will be computed, and this value will be assigned as the approximation for a feature. To complete the pair-wise comparisons between all products, at least $2 \cdot n$ measurements are needed. By using the values of previous measurements, the number of required measurements is reduced to $n + 1$.

Based on the results of case studies, the authors argue that this approach is able to improve the accuracy of prediction (over 99% if the feature relationships are known), and also reduces measurement efforts significantly ($2^n$ for measuring $n^2$ products in the worst case).

3.2.2 F-SIG

As quality attributes are critical issues in software development, many efforts have been made to measure the degree to which a software system satisfies its quality attributes. Jarzabek et al. (2006) proposed an approach to addressing quality attributes in SPLs by integrating FODA (Kang et al., 1990) and Goal-oriented analysis (Chung et al., 2000; Mylopoulos et al., 2001). The integrated framework improves both research and practice in the product line context.

FODA is a well-known domain analysis method of modelling the requirements in SPLs, focusing mainly on the dependency relationships among variant features. Goal-oriented analysis, on the other hand, aims to represent and analyse the implicit relationships between features and quality attributes. In the product-line architecture design phase, the integrated framework provides design rationale by addressing the interrelationships between features and quality attributes. During the construction, developers are able to evaluate the impact of feature selections onto system quality based on their interdepen-
dencies identified. The integrated modelling notation is named “Feature-Softgoal Inter-
dependency Graph” (F-SIG), which is used to evaluate the impact of a feature’s selection
on the final product’s quality attributes, either positively or negatively. The relationship-
s included in F-SIG are mainly categorised as structural interdependencies and implicit
interdependencies. The intention of categorising relationships is to better represent the
contributions and constraints between functional features and quality attributes. The
detailed descriptions of relationship categories are available in Jarzabek et al. (2006).

In this approach, F-SIG diagrams are used instead of conventional feature models.
These diagrams are further split into packages to deal with a large amount of product
lines data, also providing better visualisation of interdependencies. Packages are easily
structured to represent different parts of an F-SIG diagram, and refer to many places.
The adaptation of packages is essential for scalability of the whole F-SIG diagram.

3.2.3 Quality Prediction for Product Lines

Zhang et al. (2003) also propose an approach to model and represent explicitly the impact
of variants, e.g. configuration decisions, on features and quality attributes. The approach
is based on a Bayesian Belief Network (BBN) (Jensen, 1996) to help quality prediction and
assessment for an SPL. BBN provides a graphical model which can be used to represent
variants, and relationships between variants, in terms of nodes and directed edges.

BBN-based approach adapts domain experts’ knowledge, and experience from previous
similar projects, to reflect their beliefs in how much variants influence related quality
attributes. It is helpful to capture the impact of variants, and assess product qualities by
performing quantitative analysis. As the impact of variants on software quality is hardly
identified accurately, this approach uses the concept of probability to evaluate whether
systems have high or low quality. The concept of probability represents the risks which
could be involved in the configuration decisions.

In the BBN model, desired features and quality attributes are considered; the configuration decisions that contain various combined features are listed, and the influential relationships between features and quality attributes are also identified. To assess the impact of design decisions, different probability-ranges are assigned to reflect the degrees of influence. For example, high influence ranges from 0.7 to 1.0, medium from 0.4 to 0.6, and low from 0.0 to 0.3. After all nodes in the BBN model have been quantified, the next step is to perform quantitative analysis to predict the quality of the target system. Since we have a potential target product in mind, we are able to predict its quality by propagating configuration decisions and updating the probability distributions of related nodes (features) in the BBN graph.

Finally, it suggests that to have more precise evaluation, more rigorous techniques such as the Analytical Hierarchy Process (AHP) (Saaty, 1980) should be adopted by assigning weights to each quality attribute and calculating an overall score for each configuration decision.

3.2.4 Summary

Quality-oriented evaluation for product lines focuses on investigating and identifying the contributions or relationships between functional features and quality attributes. Instead of actually generating products, these approaches implement the concept of predicting and assessing the quality of intended target systems.

Because of the complexity in SPLE, it is impossible to address all quality-related issues within a single approach or framework. There are limitations to these approaches, which require further modifications and incorporation with more rigorous techniques. For example, the representation of interrelationships between features and quality attributes...
needs to be enhanced, or quantitative analysis support should be improved.

3.3 AHP-based Approach

To understand the improvements achieved by our approach, it is necessary to understand the AHP approach and the open problems we are dealing with. In the approach by Zhang et al. (2010), the NFRs framework (Chung et al., 2000) is used to identify all the critical quality attributes for an SPL and extend conventional feature models by including these quality attributes within a sub-feature tree. Then the AHP comparison process will be conducted to evaluate the relative impacts of two features on a quality attribute based on the domain expert’s judgments. The domain expert will assign a higher score to the more important features, i.e. the feature which has greater impact on the corresponding quality attribute. Finally, all comparison results given by each domain expert are put together, and a ranking is produced for all the features involved, based on a parameter called “Relative Importance Value” (RIV), which represents a feature’s overall contributing weight for related quality attributes.

In this approach, the comparison results rely totally on domain experts’ judgements. This may introduce some errors, such as inconsistency issues. For example, for satisfying a particular quality attribute, a domain expert ranks feature A as more important than feature B and feature B as more important than feature C. Suppose, if later the domain expert ranks feature C as more important than feature A, then we have an inconsistency in the ranking. This kind of error can be identified by checking the consistency ratio (CR) of the comparison matrix. AHP allows for small inconsistencies in judgments, because human judgments are not always consistent, especially when the feature model is large. Another issue is that rankings based on different domain experts’ assessments are not likely to follow a common trend. If there are obvious differences existing between several
rankings, it is difficult to tell which one is more accurate.

To address these deficiencies, we have developed an improved approach by adapting the Elo rating system (Elo, 1978) to enhance quality-oriented product configuration.

3.4 The Elo Rating System

The Elo rating system (Elo, 1978) was developed in the late 1950s and named after its creator, Professor Arpad Elo. It was originally developed to calculate the relative skill levels of two competing chess players and was adopted by the International Chess Federation (FIDE) in 1970. Based on pair-wise comparison, the Elo rating system has been widely adopted and used by various associations in today’s sports rankings such as bridge, tennis, baseball, football, rugby, even video game ranking etc.

The Elo rating system assigns each player a numerical rating, based on their performance in competitions. In original chess ratings, a player’s rating is normally a number between 0 and 3000. This rating changes over time, depending on the outcomes of tournament games. When two players compete against each other, the player with the higher rating is expected to win more frequently than his opponent. The greater the gap between two competitors, the greater the predicted likelihood that the higher-rated player will win the game.

There are two critical statistical components to the Elo rating: 1) calculating the expected score \((E)\) for a player, and 2) updating players’ ratings. A player’s expected score \((E)\) is his probability of winning, plus half his probability of a draw. Consider two players A and B; if player A has true strength \(R_A\) and player B has true strength \(R_B\), the formula to calculate the expected score of Player A is:

\[
E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}
\]

Similarly, the expected score for Player B is:
\[ E_B = \frac{1}{1 + 10^{(R_A - R_B)/400}} \]

Note that, since a player’s true strength is unknown, the expected score is calculated by using the player’s current rating in practice. Since a player’s expected score represents the proportion of his winning opportunity against his opponent, \( E_A + E_B = 1 \). For the initial rating, all players are assumed to have the same ratings. So for any player, the expected score for the first game is \( E = 0.5 \), as \( R_A = R_B \).

To update their ratings, players’ expected scores and actual scores need to be taken into account. The formula for adjusting a pre-tournament rating is:

\[ r_{post} = r_{pre} + K(S - S_{exp}) \]

where \( r_{post} \) is a player’s updated rating, \( r_{pre} \) is a player’s rating before a tournament. \( S \) represents a player’s actual total score in a tournament. For example, in a chess game, a player will be assigned 1 point if he wins, 0.5 if he draws, and 0 if he loses. \( S_{exp} \) represents the total of a player’s expected scores against all his opponents, e.g. \( S_{exp} = E_1 + E_2 + \ldots + E_n \), where \( n \) is the numbers of games he played in a tournament. \( K \) is an attenuation factor that determines the weight that should be given to a player’s performance relative to his pre-tournament rating. For chess players, \( K \) is normally set at 16 for masters and 32 for weaker players.

Updating the ratings could occur after each game, or after a tournament of several rounds. An example will be helpful in understanding how the Elo system works. Suppose that a player, who is rated at 1500, played four games in a tournament. He lost to a player rated at 1450, drew with a player rated at 1520, won against a player rated at 1475, and beat another player rated at 1600. His actual score was \((0 + 0.5 + 1 + 1) = 2.5\) . His expected score, calculated by using the first formula, was \((0.571 + 0.471 + 0.536 + 0.36) = 1.939\). Then his new rating would be \(1500 + 32 \times (2.5 - 1.939) = 1518\), assuming that a \( K \) factor of 32 was used. So this player’s rating increased as he drew with a higher-rated
opponent, defeated another higher-rated player, and lost to a lower-rated player. His overall performance in this tournament was better than he had been expected according to the Elo rating system.

3.4.1 Approach Overview

In our approach, the main focus is to clarify the relationships between quality attributes and features. For each quality attribute of the final product, we identify a list of features from the feature model which might make some contribution towards satisfying the corresponding quality attribute. Then each feature is assigned the same initial rating. The domain experts will randomly select two features under the same quality attribute and evaluate the relative importance of these two features. As domain experts only provide a relative ranking between two features, there will not be any critical mistake if one particular feature is misunderstood. Similar to chess players’ ratings, features can also be rated and ranked after some rounds of pair-wise comparisons.

Our approach contains six main steps. These steps are designed to identify the impact of features on various quality attributes of a software product, and to conduct the comparisons between features:

- **Step 1**: The first step is to identify all the critical quality attributes for a product family. The NFR framework (Chung et al., 2000) could be used to support quality attributes identification. These identified quality attributes should then be subdivided into more detailed sub-quality factors.

- **Step 2**: The goal of this step is to relate features to corresponding quality attributes based on their contributions. Quality attributes and their contributing features should be identified as groups. This step is mainly conducted based on domain experts’ knowledge and experience.
• **Step 3:** The third step is conducted by domain experts. As mentioned, domain experts should compare a randomly selected pair of features and identify which one is more important in terms of satisfying a given quality attribute. In this step, domain experts should not consider the existing ranking of the features and only conduct comparisons based on their own experiences. Similar to a chess game, one competitor (feature) will be given a certain value for each comparison against his (its) opponent (another feature).

• **Step 4:** In this step, each competitor feature’s expected score for each comparison against the opponent should be calculated by using the first formula of the Elo rating algorithm:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A) / 400}}$$

in which $R_B$ represents the opponent’s current rating, and $R_A$ represents feature A’s current rating. Assume that feature A is involved in 5 comparisons against different features. There should be 5 values of $E_A$ corresponding to each comparison.

• **Step 5:** The fifth step is to update the feature ratings after the comparisons by each domain expert are completed, noting that the number of comparisons might be different for different domain experts. To update feature ratings, the second formula is used.

$$r_{post} = r_{pre} + K(S - S_{exp}),$$

in which $r_{post}$ represents the new rating, and $r_{pre}$ represents the current rating. $S$ represents the total values actually assigned to the feature, and $S_{exp}$ represents the sum of expected scores for all the comparisons involved.

Steps 3-5 should be repeated to update features’ ratings from all domain experts available.
• **Step 6:** The final step is to incorporate various rating lists and produce a comprehensive chart of features’ contributions to different quality attributes, e.g. one feature has an impact on several quality attributes. This step demonstrates how the results produced by our approach could be used in quality-oriented product configuration.

Using our approach, we will eventually get several feature lists. Each list indicates the relative importance of a feature set to a related quality attribute. These feature lists can be used to provide a general idea of features’ contributions and guide product configurations by selecting appropriate features to achieve required quality attributes. Moreover, in the instance that one feature contributes to several quality attributes, the rankings of the feature for different quality attributes are able to guide tradeoff issues in configuration.

As we have mentioned before, when a feature model contains a lot of features, it is time-consuming for the domain experts to provide a comparison for every pair of features related to a quality attribute. It is more practical for domain experts to provide partial comparisons, where only some of the features are compared. It is not difficult to see that such situations can be easily integrated into our approach. Our approach is tolerant, to a certain degree, of inconsistent ratings from domain experts, thus avoiding the process of identifying and fixing any inconsistent decisions. One more thing to note is that, in the Elo algorithm, if a player played and won more games, then his rating was higher than those who played fewer games or won fewer games. So our approach has the same effect: if a feature has been compared more often, then it has a higher rating. This might looks unreasonable at first glance, however, when such a feature is compared with another features and loses in the comparison, it is going to help the other features to gain a greater advantage in ratings based on the Elo algorithm. This maintains the integrity of the ranking list. Also, if a domain expert ranks two features differently to the previous
rankings given by other domain experts, the ratings of these two features will be affected more obviously than the case in which the comparison result of the two features is same as others, because the comparison result is different from the expected one. In other words, our approach is very sensitive to anomalies: it amplifies the opposite opinions to make sure that no important information is missed.

3.4.2 Case Study

To illustrate our approach, we have conducted a case study using the Computer Aided Dispatch (CAD) software product line (Zhang et al., 2003) for both AHP and the Elo approach. By comparing both approaches, we intend to show how the Elo approach manages the issues which exist in the AHP approach.

The case study used includes a set of mission-critical systems which is used by police, fire rescue, health service and many other users. We have established a feature model for the CAD systems and also identified several quality attributes with contributing features. Table 3.1 represents the identified quality attributes with related features.

<table>
<thead>
<tr>
<th>Table 3.1: Quality Attributes and Contributing Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of Use</td>
</tr>
<tr>
<td>Command-Line</td>
</tr>
<tr>
<td>Drag-Drop</td>
</tr>
<tr>
<td>Colour-Based</td>
</tr>
<tr>
<td>TimerAlerts</td>
</tr>
<tr>
<td>One-Login</td>
</tr>
<tr>
<td>CallGuide</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

3.4.2.1 AHP Approach

In this section, we present the experiment results of the AHP approach. For illustrative purposes, we only present the result involving the quality attribute “Ease of Use” (EU)
and its contributors (features). We use results from two (out of five) domain experts to highlight the process and the issues with the AHP approach. In this experiment, we used the AHP support tool developed by BPMSG (2012).

Before we show the results of experiments, we need to understand the term - “Relevant Values” (RV) defined in the AHP model. The relevant values range from 1 to 9. If two features are considered to contribute equally to the objective, then a value of 1 will be assigned to each feature. If experience and judgment strongly favour A over B, then a value of 5 will be assigned to A, while a value of 1/5 will be assigned to B. The detailed definition and description of RV is defined by Saaty (2008).

Since we have assigned values to each feature when comparing each pair of features, we are able to work out a relative importance value (RIV) for every feature to represent how important a feature is as a contributor, compared to other features. Based on their RIVs, we can easily see which feature in the set has a greater contribution to the achievement of the related quality attribute. We also need to check the consistency ratio (CR) from the results to decide if the result is acceptable. We rely on the AHP tool (BPMSG, 2012) to calculate both RIVs and CR.

Table 3.2 shows the relative importance matrix for each pair of features on “Ease of Use” compared by the first domain expert. For each comparison, a value is assigned to the corresponding cell. For example, the domain expert believes that “Drag-Drop” is strongly more important than “TimeAlerts”, so a value of 5 is assigned to the corresponding cell of the matrix, and a value of 1/5 is assigned to “TimeAlerts”. “Command-Line” is moderately less important than “Colour-Based”, so a value of 1/3 is assigned. The calculation results of RIVs and the rank of all features also appear in the last two columns of the matrix.

The calculated CR represents the comparison conflicts involved. In the case of Table
Table 3.2: Relative Importance Matrix given by the First Domain Expert

<table>
<thead>
<tr>
<th>Ease of Use</th>
<th>Command-Line</th>
<th>Drag-Drop</th>
<th>Colour-Based</th>
<th>TimeAlerts</th>
<th>One-Login</th>
<th>CallGuide</th>
<th>RIV</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line</td>
<td>1</td>
<td>1/5</td>
<td>1/3</td>
<td>1</td>
<td>1/6</td>
<td>2</td>
<td>7.3</td>
<td>6</td>
</tr>
<tr>
<td>Drag-Drop</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1/3</td>
<td>3</td>
<td>23.8</td>
<td>2</td>
</tr>
<tr>
<td>Colour-Based</td>
<td>3</td>
<td>1/3</td>
<td>1</td>
<td>5</td>
<td>1/5</td>
<td>1</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>TimeAlerts</td>
<td>1</td>
<td>1/5</td>
<td>1/5</td>
<td>1</td>
<td>1/3</td>
<td>3</td>
<td>9.1</td>
<td>5</td>
</tr>
<tr>
<td>One-Login</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1/3</td>
<td>30.8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CallGuide</td>
<td>1/2</td>
<td>1/3</td>
<td>1</td>
<td>1/3</td>
<td>3</td>
<td>15.9</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

3.2, consistency ratio $CR = 0.382$. As mentioned, only for a comparison matrix, a $CR$ below 0.1 is acceptable, so this comparison result has to be rejected, and the domain expert has to redo the comparison process. Using the supporting tool by BPMSG (2012), it is easy for us to detect these conflicts and fix them. For Table 3.2, a critical conflict arises because all features are greater than or equal to “CallGuide” except “One-Login”, while “One-Login” is greater than all the other four features.

Table 3.3 shows the adjusted results from the same domain expert with $CR = 0.077$, which is an acceptable result. Comparing this with Table 3.2, we can see that the RIVs of the last three features have changed dramatically, and so have the rankings of features. The biggest change happens between “One-Login” and “CallGuide” from 1/3 to 8. So in the AHP method, consistency issues should take priority in consideration. Note that with $CR = 0.077$, there are still some conflicts within this matrix, however it is acceptable under the AHP model.

Table 3.3: Fixed Matrix given by the First Domain Expert

<table>
<thead>
<tr>
<th>Ease of Use</th>
<th>Command-Line</th>
<th>Drag-Drop</th>
<th>Colour-Based</th>
<th>TimeAlerts</th>
<th>One-Login</th>
<th>CallGuide</th>
<th>RIV</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line</td>
<td>1</td>
<td>1/5</td>
<td>1/3</td>
<td>1</td>
<td>1/6</td>
<td>2</td>
<td>6.2</td>
<td>5</td>
</tr>
<tr>
<td>Drag-Drop</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1/3</td>
<td>3</td>
<td>23.5</td>
<td>2</td>
</tr>
<tr>
<td>Colour-Based</td>
<td>3</td>
<td>1/3</td>
<td>1</td>
<td>5</td>
<td>1/5</td>
<td>3</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>TimeAlerts</td>
<td>1</td>
<td>1/5</td>
<td>1/5</td>
<td>1</td>
<td>1/3</td>
<td>1/3</td>
<td>4.5</td>
<td>6</td>
</tr>
<tr>
<td>One-Login</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>45.5</td>
<td>1</td>
</tr>
<tr>
<td>CallGuide</td>
<td>1/2</td>
<td>1/3</td>
<td>1/3</td>
<td>3</td>
<td>1/9</td>
<td>1</td>
<td>6.3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.4 shows the matrix from the second domain expert. Table 3.4 is also an acceptable result with $CR = 0.073$. However, compared to Table 3.3, the ranking of
features is quite different. It is hard to tell which one is better or more accurate. Using the AHP method, it is hard to compromise and incorporate various opinions. This is a critical drawback of the AHP method.

### 3.4.2.2 The Elo Approach

We now conduct another experiment on the same example using the Elo approach, and show how it overcomes the drawbacks of the AHP method. We have invited five domain experts to conduct the experiment by comparing the relative contributions of the feature to satisfying the given quality attributes. Just as with the chess players, a feature will be assigned 1 point if it wins, 0.5 if it is a draw, and 0 if it is considered less important. Each feature is assigned 100 points as a default rating and the $K$ factor is initially set to 10, which represents the maximum points which could be assigned in a single comparison. Note that two of the five domain experts also conducted the previous case study with the AHP approach.

Table 3.4: Relative Importance Matrix given by the Second Domain Expert

<table>
<thead>
<tr>
<th>Ease of Use</th>
<th>Command-Line</th>
<th>Drag-Drop</th>
<th>Colour-Based</th>
<th>TimeAlerts</th>
<th>One-Login</th>
<th>CallGuide</th>
<th>$RIV$</th>
<th>$Rk$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line</td>
<td>1</td>
<td>1/4</td>
<td>1/2</td>
<td>1</td>
<td>1/3</td>
<td>2</td>
<td>8.3</td>
<td>5</td>
</tr>
<tr>
<td>Drag-Drop</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>34.6</td>
<td>1</td>
</tr>
<tr>
<td>Colour-Based</td>
<td>2</td>
<td>1/3</td>
<td>1</td>
<td>3</td>
<td>1/3</td>
<td>4</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>TimeAlerts</td>
<td>1</td>
<td>1/4</td>
<td>1/3</td>
<td>1</td>
<td>1/2</td>
<td>5</td>
<td>10.7</td>
<td>4</td>
</tr>
<tr>
<td>One-Login</td>
<td>3</td>
<td>1/2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>25.6</td>
<td>2</td>
</tr>
<tr>
<td>CallGuide</td>
<td>1/2</td>
<td>1/3</td>
<td>1/4</td>
<td>1/5</td>
<td>1/6</td>
<td>1</td>
<td>4.8</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3.5: Feature Comparisons given by the First Domain Expert

<table>
<thead>
<tr>
<th>Ease of Use</th>
<th>Command-Line</th>
<th>Drag-Drop</th>
<th>Colour-Based</th>
<th>Time Alters</th>
<th>One-Login</th>
<th>Call Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Drag-Drop</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Colour-Based</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Time Alters</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>One-Login</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Call Guide</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>0.5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4.5</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 3.5 shows the comparison results given by the first domain expert, and the actual total scores ($S$) gained by each feature are summed at the bottom of the table. Based on the opinion from the domain expert, each feature’s rating and its rank will be calculated by using the Elo algorithm. All features received the same default rating on initialisation, and all the comparisons are expected as a draw, with expected score $E = 0.5$. From Table 3.5, we can see that each feature is involved in five comparisons, so the expected total score $S_{exp} = 2.5$. Comparing the $S$ achieved, some features are considered as more important contributors, e.g. “One-Logging” and “Colour-Based”. And some other features are less important, such as “Command-Line” and “TimeAlerts”.

Table 3.6 shows the corresponding changes in features’ ratings, based on the comparison given by the first domain expert. We can see that “One-Login” gained the highest score, so it has achieved the highest rating in the feature rank, where the rating is calculated as

$$r_{post} = 100 + 10 \times (4.5 - 2.5) = 120$$

<table>
<thead>
<tr>
<th>Ease of Use</th>
<th>$S$</th>
<th>Rating</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line</td>
<td>0.5</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>Drag-Drop</td>
<td>3</td>
<td>105</td>
<td>3</td>
</tr>
<tr>
<td>Colour-Based</td>
<td>4</td>
<td>115</td>
<td>2</td>
</tr>
<tr>
<td>Time Alerts</td>
<td>1</td>
<td>85</td>
<td>5</td>
</tr>
<tr>
<td>One-Login</td>
<td>4.5</td>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>Call Guide</td>
<td>2</td>
<td>95</td>
<td>4</td>
</tr>
</tbody>
</table>

To reduce the calculation, we take all the comparisons given by each domain expert as a round. Our Elo algorithm will first update the feature ratings at the end of a round, then re-rank all the features. In the following round, the ratings of the features will be used to calculate the expected score, which will be used to calculate the new ratings. Fig. 3.1 shows the change of the ratings for all related features of the quality attribute “Ease
of Use” after five rounds of comparisons. We would like to emphasise that the Elo is an immediate updating method, and we could also update feature ratings after each single comparison. However, this results in more calculations, and very often we found that there is little change in the ranking.

Based on Fig. 3.1, we can see the trend of rating change after each round. The rating of “One-Login” has been increasing all the time, which indicates that this feature has been considered as the most important feature in common by five domain experts. On the contrary, the rating of “Colour-Based” increases in the first two rounds and then decreases in the last three rounds. As mentioned, by using the Elo rating system, even if one feature is misunderstood by a domain expert, this will eventually be corrected by the results from other resources. With more comparisons, the results will be more accurate.

Fig. 3.1: Rating update for “Ease of Use”

Fig. 3.2 represents the actual score gained for features after each round of comparisons. By comparing Fig. 3.1 and 3.2, we are able to figure out the impact of actual scores on features’ ratings. For example, in the fourth round, “Command-Line” is assigned the second highest score (3), so its rating increases accordingly in Fig. 3.2. However its ranking remains in the same place (5th). So by using the Elo system, the feature ratings will remain relatively stable and will not change dramatically with a single decision. This
is consistent with previous decisions.

![Figure 3.2: Actual Scores for Each Feature of “Ease of Use”](image)

Now we would like to discuss how to integrate partial results, which cannot be done in the AHP approach. Similarly to the real world case, in which a player does not participate in all the tournaments, if a feature misses some comparisons in a round or even the whole round, the Elo algorithm will update this feature’s rating based only on the comparisons in which it has participated. So, in the extreme case where a feature misses the whole round, its rating will remain unchanged. However, its ranking could change as the ratings of other features have changed. Our Elo-based algorithm only takes features that have been compared, into account, and the missed feature will not be touched until the next time it is involved in a comparison. If a low-ranked feature wins over features with higher rankings, then it obtains a greater increase in rating compared to the case in which a higher-ranked feature wins over a lower-ranked feature. So, a ranking of a feature is relevant to the rankings of other features; thus it is not important if a feature is missed for a few rounds, as it will be properly ranked when it has been adequately compared with other features in the following rounds.

By using the Elo approach, five domain experts’ opinions are aggregated and incorporated to produce a comprehensive ranking of features. Moreover, there is no need to
consider inconsistency issues in the comparison process, because they will be corrected by
the majority vote. So it is not necessary to judge if each single result is correct or not.

3.5 Summary

Overall, our Elo-based approach is able to correlate the data from different sources, and it
is an instant system, which means that a new list can be generated whenever a comparison
result is available. Our approach is certainly an improvement over the existing approaches
in these two aspects, and will help in improving quality-oriented feature ratings. However,
we believe that there are still a few issues to be investigated further. In the Elo-based
approach, if all features have participated in the same number of comparisons, then all
features have the same chance of obtaining points or losing points in the rating; thus, the
ranking is accurate. However, if the features have participated in different numbers of
comparisons, then the features which participate in fewer comparisons have less chance
of obtaining points or losing points when calculating the rating. In an extreme case, if a
feature has not participated in any comparison, then its rating will not change at all; at
the same time, other features’ ratings will change if they are in some comparisons. This
might result in a change of ranking for the features not included in any comparison. We
would like to see how the partial rankings affect the final result, and how the Elo-based
algorithm behaves in the real-world scenario in this respect, should be investigated further.

Another topic which we would like to investigate further is a study of the impact of
the feature groups on quality attributes. In the work by Zhang et al. (2010), some efforts
have been made to define the impact of the feature groups on the quality attributes.
Basically, four kinds of groups have been defined. SumGroup, AvgGroup, MaxGroup and
MinGroup represent the sum of the RIV, average RIV, Maximum RIV and Minimum
RIV respectively. One of these should be taken as the RIV of the whole feature group.
The paper has acknowledged that domain experts should be able to define domain-specific feature groups, and we would like to investigate this further. We have introduced an aspect-oriented approach (Tan et al., 2013) to model the relationships between NFRs and functional features. This might describe more precisely the impact of the feature groups on quality attributes.

Having a large-scale case study is what we are planning. Currently, we are developing a Software Product Line based on the City Evolution Project from the Newcastle Council, and we hope this can be completed in a few months’ time.
Chapter 4

Efficient Product Configuration

4.1 Introduction

Product configuration is designed to select appropriate features from the feature model, and the selections have to satisfy stakeholders’ expectations, e.g. NFRs. Moreover, relationships and constraints between features also affect the decisions on feature selections. So minimising invalid and undesired selections in the configuration will definitely improve the overall quality of a product configuration. Disregarding quality issues, feature-based configuration has to be efficient, because in practice, feature models normally contain hundreds, even thousands of features. It is inefficient and costly to roll back and fix manually any error that has occurred during the configuration process. Thus a well-designed configuration approach should, ideally, not only be qualified, but also efficient.

To improve the efficiency of product configuration, we propose a configuration approach that deals with relationships and constraints between features (Tan et al., 2013). Features do not exist independently in a feature model. There are always relationships and constraints between features, e.g. “requires” and “excludes”. When configuring a product, we need to traverse the feature model and make a configuration decision at each
variation point to select variant(s). In conventional product configuration, there is no fixed order or sequence, so configuration could start from any hierarchy of the feature model. Usually a depth-first traversal will be employed. Assuming that there are two variation points, VP1 and VP2, we first encounter VP1 during the traversal and select a variant at VP1. If the selected variant at VP1 “requires” a variant at VP2, then the required variant at VP2 has to be selected based on the “requires” dependency. Thus, we do not need to visit this variant at VP2 later. In this case a selection of variants at one variation point may already cover the selections at other variation points. If those variation points with greater coverage are processed first, obviously, there will be fewer decisions to make, thus the configuration process is more efficient. Now, assume that we make a configuration decision at VP2 first, and mistakenly decide not to include this variant in the final product. We will not realise that this is a wrong decision, until we make a configuration decision at VP1. In this case, we have to go back to VP2 again to change previous decisions. In some situations, the corrections might propagate, and thus become time-consuming to fix. Thus, in terms of configuration efficiency and quality, it is better to visit VP1 first in the configuration.

So the sequence of the variation points following which we make our configuration decisions has a significant impact on the efficiency of product configuration. Getting the correct sequence is a key consideration. We have proposed an approach to improve the efficiency of product configuration by taking into account feature dependencies and identifying a small set of variation points from a feature model. By making selections from this set, we predetermine all the variants in the feature model; thus we have reduced the number of variants to visit during the configuration process. As a result, the number of decisions to be made is reduced, which implies that the rollbacks caused by feature conflicts are reduced.
Note that, in the current approach, we only include “requires” and “excludes” relationships. We would like to point out that our approach is extendable to cover other types of dependencies, if they are defined accurately and the logical operations involving these dependencies are defined precisely.

4.2 Literature Review

The literature review for product configuration covers two aspects. One is feature model analysis, which provides solid and valid feature models with clearly identified feature relationships for configuration. The other is feature-based product configuration, based on the main drivers of configuration considerations.

4.2.1 Feature Model Analysis

4.2.1.1 Optimised Feature Model Verification

Yan et al. (2009) proposes an approach to improving feature model verification. It suggests that, to improve the efficiency of verification, the problem should be optimised before involving third-party tools. The goal of feature model verification is to remove the conflicts in constraints among features. For large feature models, it is not feasible to conduct verification without any optimisation strategy. This approach also includes some criteria for feature model verification, and all the anomaly and inconsistency deficiencies will be detected according to these criteria.

The strategy represented in this approach is mainly based on the observation of feature models which contain some features and constraints that are irrelevant to the feature models’ verification. The main intent is to remove these irrelevant features and constraints from the feature model, without violating any valid constraints. For example, if child features in hierarchial relationships do not have any cross-tree relationships with other
features, these child features and their hierarchial relationships can be removed temporarily from the feature model. This simplifies the feature model verification processes and reduces the complexity of the feature model.

To sum up, all irrelevant features are those that do not appear either as themselves or as offspring, in any explicit constraint (cross-tree relationship). The constraints that involve at least one verification-irrelevant feature are also verification-irrelevant. According to the optimisation strategy, eliminating these features and constraints from the feature model will not influence its verification result.

4.2.1.2 Automated Analysis of Feature Models

Similarly, Segura (2008) has proposed an approach which focuses on simplifying feature models to improve the performance of feature model analysis. The main purposes of feature model analysis is to extract the information that shows when a feature model is void, when it contains errors, or the number of products that it represents.

This approach aims at finding atomic sets which represent a group of features that can be treated as a unit for automated analysis purpose. An atomic set of features is a combination of features and their parents, which can be treated as a whole in feature model analysis. The intention of creating an atomic set is that mandatory features and their parent features in hierarchial relationships can be treated as a whole in feature model analysis. This is because these features always exist in any member product as a feature set. Using atomic sets instead of individual features will reduce the number of variants to consider and the size of problem to deal with. The approach has proven that it improves the performance of feature model analysis, not only in large feature models, but also in small cases.
4.2.1.3 Summary

Feature model validation is conducted to examine the correctness and completeness of a feature model. A verified feature model is expected to provide a conflict-free basis for product configuration later on. However, there are some limitations remaining in the approaches above. For example, only mandatory features and implicit constraints are considered in validation approaches. Explicit feature relationships are not covered. As a result, in feature models with large numbers of cross-tree relationships, the efficiency will be reduced.

4.2.2 Product Configuration Approaches

4.2.2.1 Multi-step Configuration

As well as conventional product configuration, there is another method, which is known as multi-step configuration. The multi-step configuration is the process of gradually configuring a product from an SPL, adding features to the configuration step-by-step. The multi-step configuration corresponds to a case such as one in which the development is constrained by an annual development budget, so that new features have to be added to the products year-by-year.

White et al. (2009) has proposed a multi-step configuration approach, called MUlti-step Software Configuration probLEm solver (MUSCLE), to transform multi-step configuration problems into CSPs (Hentenryck, 1989). Then MUSCLE uses an automated tool, which is basically a CSP solver, to derive the sequence of features to be included in the multi-step development. The main objective here is to make sure that the features required by other features will be included in the early product revisions to shorten the product evolution. MUSCLE considers the constraints between features, and also takes into account the intermediate configuration constraints.
This approach can also be used for single-step configuration, in which the maximum number of configuration steps is $K = 1$. In this case, the approach has to produce a valid configuration based on given constraints using a CSP solver, which might not be the most efficient way to go.

### 4.2.2.2 Priority-based Feature Configuration

Bagheri et al. (2010) proposed a configuration approach which ranks and selects the most relevant features from the feature models in a staged configuration process, based on the system concerns or business objectives. In this approach, feature models are extended by designating one or more concerns in business to each feature. Examples of concerns are: cost, time, customer importance, and penalties. During the configuration process, these concerns will be taken as strategic objectives of the application domain and stakeholders. The Analytic Hierarchy Process (AHP) (Saaty, 1980), which is a well-known pair-wise comparison method, is employed to compute how much a feature contributes to satisfy business objectives. In the process of product configuration, the business goals and stakeholders’ needs are taken as the main considerations, and the features are selected based on their ranking according to the desired business objectives.

One of the disadvantages of this type of goal-oriented configuration is that there are many conflicts among different stakeholders’ business objectives, and resolving these conflicts introduces a lot of modifications to the configurations, which makes the process hard to follow.

### 4.2.2.3 Usage Context-driven Feature Selection

Lee and Kang (2010) proposed an approach for feature selection in configuration which considers the context in which the product will be used as a key driver. The approach
describes how usage contexts are used and related to other typical factors that significantly impact on feature selection, e.g. quality attributes and technical constraints. A domain knowledge model is developed during domain engineering to relate usage contexts to features, and this model will be used in application engineering to derive an optimal product configuration.

According to Lee and Kang (2010), usage contexts are informally defined as “any contextual settings in which a product is deployed or used, which can be detailed in terms of user, physical, social, business, operating environments, etc.” Different detailed contexts provide different data for feature selection. For example, the user context provides information used in determining quality attributes. Physical contexts present physical environments or locations in which a product is deployed and used. Some constraints of the products could be identified on the basis of this information. The benefit of analysing the product-usage contexts is that product quality attributes and technical constraints will be identified, which will further guide the selection of desired features for a product.

Once a domain knowledge model has been established, usage contexts are related to feature selections factors through multiple mappings. Then this information will be used as inputs to application engineering. Product feature selection will be conducted based on this information, and the output of the process will be a product feature set, which includes features required to satisfy the quality attributes and technical constraints specified in the domain knowledge model.

4.2.3 Summary

In practice, configuration decisions are often constrained by some explicit concerns and constraints, such as budget, cost, time and technical requirements. All the configuration approaches represented take these concerns as key drivers and deal with them by imple-
menting some extra techniques and support tools. By prioritising and filtering features based on business goals and objectives, the efficiency of configuration is improved and the quality of the configured products is preserved.

Apart from explicit concerns, implicit feature relationships should also be considered to improve the efficiency of the product configuration. This consideration should be adopted in multiple configuration approaches in order to effect a more comprehensive sequence of actions to guide feature selections.

4.3 Approach Overview

The sequence of the variation points can be determined based on a parameter which we call \textit{Configuration Coverage}. Configuration Coverage (CC) of a variation point describes the extent to which a configuration decision, made at a variation point, covers the configuration decisions of the remaining variation points in a feature model. To improve the efficiency of the configuration process, it is crucial to identify a minimal set of variation points, in which the decisions made at this set of variation points cover the decisions to be made at all the variation points in a feature model. This set of variation points will be sorted based on their configuration coverage, and we will start making configuration decisions at the variation point with the largest configuration coverage. However, as the feature model could be very complex, and the feature dependency model could be hard to trace, it is not always straightforward to find such a minimal set of variation points for a product configuration. We propose employing some well-studied mathematical techniques to help to identify the minimal variation point set from a feature model. Before we present the proposed approach, we will first define some measurements used in the approach.

In a feature model, each variation point consists of a set of variants and a multiplicity. For a variant \( v \), we define two measurements. One is called the \textit{Positive Coverage} \( PC(v) \),
the other is called the *Negative Coverage* $NC(v)$. When the variant $v$ is included in a product configuration, the positive coverage of variant $v$ ($PC(v)$) is a set of variable features which will be automatically included or excluded, based on their dependence relationships within variant $v$. Similarly, when the variant $v$ is excluded from a product configuration, the negative coverage of variant $v$ ($NC(v)$) is a set of variable features which will be automatically included or excluded, based on the multiplicity constraints between these variants. To work out the positive and negative coverage, we need to examine the dependency relationships among the variants in a feature model. For example, if variant $v$ requires variant $w$, then we know that $w$ is in $PC(v)$. Furthermore, if $w$ requires variant $u$, then $u$ is also in $PC(v)$ since if $v$ is included in the final product, then $u$ will be included as well. Turning to negative coverage, if variant $t$ requires variant $v$, then we know that $t$ is in $NC(v)$, since if $v$ is not included in the final product, then $t$ cannot be included as well.

For a variation point in a feature model, the multiplication rule restricts the selection of the variants associated with the variation point. For example, a multiplicity of $1..n$ means only up to $n$ variants can be selected in the final product. For all the variants associated with a variation point, we call a subset of variants a *valid selection* if it obeys the multiplicity parameter. The *complement* of a valid selection is the set of variants that are not included in the selection at the variation point. When a certain valid selection has been made at a variation point, the configuration coverage of the selection is the union of all the positive coverage of the variants in the valid selection and all the negative coverage of the variants in the complement of the selection. Below is an example to illustrate how to calculate these parameters. Note that a variation point may have different configuration coverage when different valid selections are made at the variation point.

Included in Fig. 4.1 is a fraction of a feature model. There are four variants $v1$, $v2$,
Figure 4.1: A Variation Point (VP) and its Variants

$v3$ and $v4$, associated with the variation point $VP$. The multiplicity parameter restricts the selection, i.e. only up to two variants can be included in the final product. From the dependency relationship of Fig. 4.2, we know that:

\[
PC(v1) = \{u1, u3\}, \quad NC(v1) = \emptyset, \quad PC(v2) = \{u2, u4, u7, u8\}, \quad NC(v2) = \emptyset, \quad PC(v3) = \{u5\}, \quad NC(v3) = \{u6\}, \quad PC(v4) = \{u4, u8\} \quad \text{and} \quad NC(v4) = \{u6\}.
\]

Figure 4.2: The Dependencies among Variants

All possible selections based on the multiplicity at the $VP$ are listed below, $v1$, $v2$, $v3$, $v4$, $v1 \cup v2$, $v1 \cup v3$, $v1 \cup v4$, $v2 \cup v3$, $v2 \cup v4$, $v3 \cup v4$.

The configuration coverage ($CC$) of each selection is listed as follows:

\[
CC(v1) = PC(v1) \cup NC(v2) \cup NC(v3) \cup NC(v4) = \{u1, u3, u6\},
\]

\[
CC(v2) = \{u2, u4, u7, u8, u6\},
\]

\[
CC(v3) = \{u5, u6\}, \quad CC(v4) = \{u4, u8, u6\},
\]

62
\[ CC(v_1 \cup v_2) = PC(v_1) \cup PC(v_2) \cup NC(v_3) \cup NC(v_4) \]

\[ = \{u_1, u_2, u_3, u_4, u_7, u_8, u_6\}, \]

\[ CC(v_1 \cup v_3) = \{u_1, u_3, u_5, u_6\}, \]

\[ CC(v_1 \cup v_4) = \{u_1, u_3, u_4, u_6\}, \]

\[ CC(v_2 \cup v_3) = \{u_2, u_4, u_7, u_8, u_5, u_6\}, \]

\[ CC(v_2 \cup v_4) = \{u_2, u_4, u_7, u_8, u_6\}, \]

\[ CC(v_3 \cup v_4) = \{u_5, u_6\}. \]

The largest feature set from the above is the \( CC \) when \( v_1 \) and \( v_2 \) are selected at this variation point. We call this \textit{Maximum Configuration Coverage (MAXCC)}. The Configuration Coverage of a variation point is the MAXCC of all the valid selections at the variation point. If a selection of variants at a variation point can result in MAXCC, we call the selection a \textit{Max Coverage Selection (MCS)}. In the above example, \( MCS = v_1 \cup v_2, MAXCC = CC(v_1 \cup v_2) = \{u_1, u_2, u_3, u_4, u_7, u_8, u_6\} \).

The MAXCC of a variation point indicates how much a decision made at the variation point covers the decisions at the other variation points of a feature model. Potentially, the bigger the coverage of a variation point, the more variant features will be included/excluded as the result of including or excluding the variation point. Therefore, it is more important to visit the variation point earlier in the configuration process. Using the MAXCC of variation points, we can construct a small set of variation points from a feature model, the union of whose MAXCC will cover all the variant features in the feature model. Software engineers should start with this set of variation points when configuring a product from the feature model. This set of variation points represents the key decisions.
for configuring a member product. Focusing on this set of variation points will reduce the configuration effort.

One thing we would like to point out is that, in the configuration process, the $MAXCC$ of a variation point might not correspond to the $CC$ of the actual selection. i.e. the actual selection might not give the $MAXCC$. The $MAXCC$ of a variation point only indicates the potential importance of the variation point in the configuration process.

Our approach works like this. We firstly work out the $MAXCC$ of every variation point in the feature model, and then calculate the smallest set of variation points that covers the whole feature model. In other words, the union of the $MAXCC$ of this set of variation points includes all the variants of the feature model. The software engineers could start from this set of variation points to configure the final product. Once a decision is made on a variation point (or a set of variation points), we then work out the coverage of the selection(s). This is a straightforward task, as we already know the positive coverage and negative coverage of each variable feature. For the rest of the features not yet covered, we will then repeat this process until all the variants are covered.

There are many ways to identify a minimal set of variation points which covers a feature model. One of the simpler, but less optimal approaches would be using a greedy algorithm, i.e. selecting the variation points with the biggest coverage until the union of the coverage covers the feature model. A more precise approach is to model the problem as the minimum vertex cover problem and then to use some approximation algorithm to solve the problem.
4.4 Vertex Cover Problem and Simulated Annealing Algorithm

To identify the minimal set of variant points for a configuration, we will employ some well known mathematical approaches in graph theory and discrete optimisation. We first need to transform a feature model into a directed graph. The transformation is quite straightforward. Every variable feature in the feature model becomes a vertex in the resulting graph, and the dependencies between two variable features become the arcs in the resulting graph. Once we can model the feature model into a directed graph, we can then apply some discrete optimisation techniques.

![Figure 4.3: Transform a Feature Model into a Directed Graph](image)

In graph theory terms, a “vertex-cover” of a directed graph (digraph) is a set of vertices such that each arc of the digraph is incident to at least one vertex of the set. A minimum vertex-cover is a vertex-cover of the smallest size. The problem of finding a minimum vertex-cover is a classical optimisation problem in computer science. This problem is a typical example of an NP-hard optimisation problem (Garey and Johnson, 1979), and an optimal solution is very hard to obtain in general. Normally, randomised algorithms become the first choice. For example, Simulated Annealing and Genetic Algorithm (Pincus, 1970) have been used very often in solving the vertex cover problem. The algorithms are efficient and can very often produce reasonably good solutions.

The problem we are dealing with here is very similar to the classical Vertex Cover problem. The key difference is that the coverage in the classical minimum vertex cover
problem is the immediate neighbour of the node, whereas in our problem, the coverage can extend to the nodes beyond the immediate neighbour. However, this difference does not introduce much difficulty into the original problem. We believe that a Simulated Annealing algorithm would still be suitable for solving our problem, as the algorithm can produce reasonably good solutions in a given time frame.

In our approach, we have implemented the HSAGA algorithm (Tang et al., 2008), which is an improvement on the SA algorithm. The algorithm combines the Genetic Algorithm (GA) and Simulate Annealing (SA) algorithm. The HSAGA algorithm has multiple iterations. In each iteration, we start from multiple instances, i.e., solutions or partial solutions, and we apply SA to these instances to produce the offspring. The offspring will then be crossed over in the GA algorithm and produce instances for next iteration. The key idea is to balance the effort between local optimisation and multiple probing. The algorithm has demonstrated its efficiency in several classical discrete optimisation problems.

We have applied the HSAGA algorithms in finding the minimal cover of the feature model. We have modified the algorithm to start from a randomly selected variable feature in the feature model, since we know its coverage, i.e., the set of vertices covered by the vertex, so we can remove the vertex and also the set of vertices it covers, and then we randomly select another vertex and repeat the same process. When this process finishes, we will fully cover the feature model. We will produce multiple coverages in the same way, and then apply the HSAGA algorithm to the instances.

We would like to note that the HSAGA algorithms do not outperform the standard SA algorithm if the instance is small, for example, if the graph has less than a few thousand nodes. If a more accurate solution is expected, then the number of iterations should be increased.
4.5 Experiments and Results

We have conducted two types of experiments on both a modified feature model (library systems) and randomly generated feature models, to evaluate our approach. The results of the experiments demonstrate that our approach is efficient.

4.5.1 Library System Feature Model

The feature model used for library systems (See Fig. 4.4) contains over 60 variable features comprising 42 variation points (VPs). Each variation point is represented by a name, such as VP1 and VP2. The dependencies among the variants are represented in Table 4.1. Each variant listed in the table is assigned a Variant ID, called V ID. The “Requires” and “Excludes” columns represent the dependencies among the variants.

Table 4.2 shows the related MAXCC of each variation point and the corresponding MCS. The MCS column in the table refers to the corresponding selection of variants that results in MAXCC for the variation point. For example, for VP7, the MCS is 4 ∪ ¬ 5, which means that if we include variant 4 but do not include variant 5, this selection of variants at VP7 will result in the inclusion/exclusion of another 4 variants from other variation points as shown in the table. This set of variants is MCS of VP7.

Based on the MAXCC of each variation point, we can find a set of variation points which cover the feature model, and the set is the smallest one possible. If the MCS of each variation point in this set is selected, then the whole feature model will be covered, i.e. we do not need to go through other variation points to define the product configuration. If in the configuration process, at a variation point, the selection is not the one giving the MAXCC, then we will have to recalculate the covering set.

A minimal covering set for the Library product line has been identified and sorted into a sequence in terms of the size of their CC, which are shown in Table 4.3. We can see that
Figure 4.4: A Feature Model of Library Systems
Table 4.1: The Dependency Relationships among Variants in a Feature Model

<table>
<thead>
<tr>
<th>VID</th>
<th>Variant</th>
<th>Requires</th>
<th>Excludes</th>
<th>VID</th>
<th>Variant</th>
<th>Requires</th>
<th>Excludes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Payment</td>
<td></td>
<td></td>
<td>31</td>
<td>Overdue Fee</td>
<td>0,2,12,17</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>eBook</td>
<td>41,51</td>
<td>48,32</td>
<td>32</td>
<td>Damage Cost</td>
<td>0,2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Fee Policy</td>
<td></td>
<td></td>
<td>33</td>
<td>Reserve Fee</td>
<td></td>
<td>0,2</td>
</tr>
<tr>
<td>3</td>
<td>Reward Policy</td>
<td></td>
<td></td>
<td>34</td>
<td>Online Reserve</td>
<td>51,57</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>Registration Fee</td>
<td>0,2</td>
<td></td>
<td>35</td>
<td>Online Cancel</td>
<td>51,57</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>Issue Library Card</td>
<td>46</td>
<td></td>
<td>36</td>
<td>Overdue Notification</td>
<td>12,17</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Replace Library Card</td>
<td>5,46</td>
<td></td>
<td>37</td>
<td>Fee Notification</td>
<td>2,12,17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Renew Fee</td>
<td>0,2</td>
<td></td>
<td>38</td>
<td>Email</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>Phone Number</td>
<td>39</td>
<td></td>
<td>39</td>
<td>Post</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Address</td>
<td>40</td>
<td></td>
<td>40</td>
<td>SMS</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Email Address</td>
<td></td>
<td></td>
<td>41</td>
<td>Digital Library</td>
<td>51,53</td>
<td>48</td>
</tr>
<tr>
<td>11</td>
<td>Reward Point</td>
<td>3,12</td>
<td></td>
<td>42</td>
<td>On-site Explore</td>
<td>51,57</td>
<td>48</td>
</tr>
<tr>
<td>12</td>
<td>Borrowing History</td>
<td></td>
<td></td>
<td>43</td>
<td>Web Explore</td>
<td>51,57</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>Update Phone Number</td>
<td>8</td>
<td></td>
<td>44</td>
<td>On-site Print</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>Update Address</td>
<td>9</td>
<td></td>
<td>45</td>
<td>Download</td>
<td>51,57</td>
<td>48</td>
</tr>
<tr>
<td>15</td>
<td>Update Email Address</td>
<td>10</td>
<td></td>
<td>46</td>
<td>Library Card Device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Loan Fee</td>
<td>0,2</td>
<td></td>
<td>47</td>
<td>Self-check Device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Record Check</td>
<td>12</td>
<td></td>
<td>48</td>
<td>Non-Network</td>
<td>1,2,24,25,26,28,29,34,35,41,42,43,44,45,49,57</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Pop-up Reminder</td>
<td>12</td>
<td></td>
<td>49</td>
<td>Network Based</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Loan Restriction</td>
<td>12,17</td>
<td></td>
<td>50</td>
<td>LAN Based</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Front Desk</td>
<td>12,17,18</td>
<td></td>
<td>51</td>
<td>Internet Based</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Self-check</td>
<td>12,17,18,47</td>
<td></td>
<td>52</td>
<td>Wireless Device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Web Search</td>
<td>51,57</td>
<td>48</td>
<td>53</td>
<td>Network Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>View Account</td>
<td>12</td>
<td></td>
<td>54</td>
<td>Message Encryption</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Website</td>
<td>51,57</td>
<td>48</td>
<td>55</td>
<td>Use Library Card</td>
<td>5,46</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>On-site Computer</td>
<td>50</td>
<td>48</td>
<td>56</td>
<td>Use Digital Certificate</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Inter-Library</td>
<td>48</td>
<td>57</td>
<td>58</td>
<td>User Web Interface</td>
<td>51,54</td>
<td>48</td>
</tr>
<tr>
<td>27</td>
<td>Onsite Loan</td>
<td>58</td>
<td></td>
<td>59</td>
<td>Credit Card</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Web Request</td>
<td>51,57</td>
<td>48</td>
<td>59</td>
<td>Digital Device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>InterLibrary Search</td>
<td>56</td>
<td>30,48,60</td>
<td>60</td>
<td>Firewall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>External Database</td>
<td>29</td>
<td>61</td>
<td></td>
<td>Proxy Server</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the variation points $V P35$, $V P9$, $V P2$, $V P8$, $V P11$, $V P10$, $V P30$, $V P15$, $V P6$, $V P27$, $V P34$, $V P31$, $V P40$, $V P41$ and $V P39$ cover all the variants of the feature model. In this particular feature model, some of the $V P$s do not have any dependency relationships with other variants, except their own parent or children variants, such as $V P4$, $V P5$, $V P26$, $V P32$ and $V P42$, which are not listed in Table 4.2. We will deal with these $V P$s after we have processed the others.

Since $V P35$ has the largest coverage in the sequence, software engineers should start the configuration by examining the variants associated with $V P35$ and making selections. Assuming that the selection of variants at $V P35$ is the same as its $M CS$, the configuration

69
Table 4.2: Max Configuration Coverage of each VP

<table>
<thead>
<tr>
<th>VPID</th>
<th>MCS</th>
<th>MAXCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP1</td>
<td>¬0</td>
<td>{4,7,16,31,32,33}</td>
</tr>
<tr>
<td>VP2</td>
<td>¬2 ∪ ¬3</td>
<td>{4,7,11,16,31,32,33,37}</td>
</tr>
<tr>
<td>VP3</td>
<td>1</td>
<td>{41,48,51}</td>
</tr>
<tr>
<td>Vp4</td>
<td>11</td>
<td>[3,12]</td>
</tr>
<tr>
<td>Vp7</td>
<td>4 ∪ ¬5</td>
<td>{0,2,6,55}</td>
</tr>
<tr>
<td>Vp8</td>
<td>6 ∪ 7</td>
<td>[0,2,5,46]</td>
</tr>
<tr>
<td>Vp9</td>
<td>¬12</td>
<td>{11,17,18,19,20,21,23,31,36,37}</td>
</tr>
<tr>
<td>Vp10</td>
<td>¬8 ∪ ¬9 ∪ ¬10</td>
<td>{13,14,15,16,31,32,33,37}</td>
</tr>
<tr>
<td>Vp11</td>
<td>13 ∪ 14 ∪ 15</td>
<td>[8,9,10]</td>
</tr>
<tr>
<td>Vp12</td>
<td>16</td>
<td>[0,2]</td>
</tr>
<tr>
<td>Vp13</td>
<td>¬17</td>
<td>{19,20,21,31,36,37}</td>
</tr>
<tr>
<td>Vp14</td>
<td>19</td>
<td>[12,17]</td>
</tr>
<tr>
<td>Vp15</td>
<td>21</td>
<td>{12,17,18,47}</td>
</tr>
<tr>
<td>Vp16</td>
<td>¬18</td>
<td>{20,21}</td>
</tr>
<tr>
<td>Vp17</td>
<td>22</td>
<td>{48,51,57}</td>
</tr>
<tr>
<td>Vp18</td>
<td>31</td>
<td>[0,2,12,17]</td>
</tr>
<tr>
<td>Vp19</td>
<td>32</td>
<td>[0,2]</td>
</tr>
<tr>
<td>Vp20</td>
<td>33</td>
<td>[0,2]</td>
</tr>
<tr>
<td>Vp21</td>
<td>34</td>
<td>[48,51,57]</td>
</tr>
</tbody>
</table>

Table 4.3: A Sequence of VPs which covers Feature Model

<table>
<thead>
<tr>
<th>VP</th>
<th>MCS</th>
<th>MAXCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP35</td>
<td>48</td>
<td>{1,2,24,25,26,28,29,34,35,41,42,43,44,45,49,57}</td>
</tr>
<tr>
<td>VP9</td>
<td>¬12</td>
<td>{11,17,18,19,20,21,23,31,36,37}</td>
</tr>
<tr>
<td>VP2</td>
<td>¬2 ∪ ¬3</td>
<td>{4,7,11,16,31,32,33,37}</td>
</tr>
<tr>
<td>Vp8</td>
<td>6 ∪ 7</td>
<td>[0,2,5,46]</td>
</tr>
<tr>
<td>Vp11</td>
<td>13 ∪ 14 ∪ 15</td>
<td>[8,9,10]</td>
</tr>
<tr>
<td>Vp10</td>
<td>¬8 ∪ ¬9 ∪ ¬10</td>
<td>{13,14,15,38,39,40}</td>
</tr>
<tr>
<td>Vp30</td>
<td>29</td>
<td>{30,48,56}</td>
</tr>
<tr>
<td>Vp15</td>
<td>21</td>
<td>{12,17,18,47}</td>
</tr>
<tr>
<td>Vp6</td>
<td>11</td>
<td>{3,12}</td>
</tr>
<tr>
<td>Vp27</td>
<td>24 ∪ 25</td>
<td>{48,50,51,57}</td>
</tr>
<tr>
<td>Vp34</td>
<td>¬46 ∪ ¬47 ∪ ¬52</td>
<td>{5,6,21,52,55}</td>
</tr>
<tr>
<td>Vp31</td>
<td>41</td>
<td>{48,51,53}</td>
</tr>
<tr>
<td>Vp40</td>
<td>55 ∪ 56</td>
<td>{5,46,59}</td>
</tr>
<tr>
<td>Vp41</td>
<td>58</td>
<td>{54}</td>
</tr>
<tr>
<td>Vp39</td>
<td>¬54</td>
<td>{58}</td>
</tr>
</tbody>
</table>
will continue to select variants from $VP_9$, that is, the second in the sequence. The configuration process will continue until all the variation points in the sequence have been visited, at which stage we attain a product configuration. This is an ideal situation, in which the $MCS$ of each variation in the sequence is selected. However, if the selected variant set at a variation point is not its $MCS$, then the covering set should be recalculated. Suppose that the selection of variants at $VP_8$ consists of 6 that are not the $MCS$ of $VP_8$. In this case, we will first work out the coverage of the new selection, which is $\{5, 46\}$, and remove this set of variants from the directed graph that we have generated at the beginning of this section. Then we will recalculate the coverage for the variant features left over. This coverage is shown in Table 4.4.

<table>
<thead>
<tr>
<th>VP</th>
<th>MCS</th>
<th>MAXCC</th>
</tr>
</thead>
</table>
| $VP_{35}$ | 48                                              | $\{1,22,24,25,26,28,29,34,
35,41,42,43,44,45,49,57\}$         |
| $VP_9$  | $\neg12$                                         | $\{11,17,18,19,20,21,23,31,36,37\}$ |
| $VP_2$  | $\neg2 \cup \neg3$                             | $\{4,7,11,16,31,32,33,37\}$      |
| $VP_{11}$ | $13 \cup 14 \cup 15$                           | $\{8,9,10\}$                      |
| $VP_{10}$ | $\neg8 \cup \neg9 \cup \neg10$                | $\{13,14,15,38,39,40\}$          |
| $VP_{30}$ | 29                                              | $\{30,48,56\}$                    |
| $VP_{15}$ | 21                                              | $\{12,17,18,47\}$                |
| $VP_6$  | 11                                              | $\{3,12\}$                        |
| $VP_{27}$ | $24 \cup 25$                                    | $\{48,50,51,57\}$                |
| $VP_{34}$ | $\neg46 \cup \neg47 \cup \neg52$              | $\{5,6,21,52,55\}$               |
| $VP_{31}$ | 41                                              | $\{48,51,53\}$                    |
| $VP_{40}$ | $55 \cup 56$                                    | $\{5,46,59\}$                    |
| $VP_{12}$ | 16                                              | $\{0,2\}$                         |
| $VP_{41}$ | 58                                              | $\{54\}$                          |
| $VP_{39}$ | $\neg54$                                         | $\{58\}$                          |

The experiment on the library system involved 20 university students. They were randomly picked and formed into two independent groups. Each group had 10 users. None of these students had any SPL experience, to guarantee that they were at the same level of expertise. The basic concept of SPL, and the purpose of the experiment, were explained to all the users. One group used a depth-first traversal method to go
through each variant and the other group used our approach. The experiment was carried out in pairs, i.e. taking one user from each group. Based on the functional and non-functional requirements listed in the requirements document, users made their selections on the feature model. A computer-based tool was provided for users to record all their decisions. When a conflict between selected features was detected by the assistive tool, users were informed by showing them where the conflict was, so that they could refine their selections. If there was anything about which a user felt unclear, then he (or she) could ask questions, and the information was shared with the peer during the experiment. This parallel process was able to provide a direct comparison of two methods. Table 4.5 shows critical parameters collected for two groups of users conducting different configuration methods (Depth-First (left), our method (right)) for library systems. Data for each pair is included in a row for comparison. Rollbacks did not happen at all for the users using our approach. From the Decision column of the two groups, we can see the gaps between two approaches are obvious, even for this simple feature model. Our approach has the advantage in terms of the number of decisions required to cover all variants, and the number of rollbacks.

Table 4.5: Depth-First Configuration Group(left) and Optimisation Configuration Group(right).

<table>
<thead>
<tr>
<th>User No</th>
<th>Decision</th>
<th>Rollback</th>
<th>User No</th>
<th>Decision</th>
<th>Rollback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>0</td>
<td>2</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>1</td>
<td>4</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>0</td>
<td>6</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>0</td>
<td>8</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>43</td>
<td>1</td>
<td>10</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>42</td>
<td>0</td>
<td>12</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>42</td>
<td>0</td>
<td>14</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>42</td>
<td>1</td>
<td>16</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>44</td>
<td>2</td>
<td>18</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>42</td>
<td>0</td>
<td>20</td>
<td>33</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.6 shows the result of average comparison of two groups (Depth-First Traversal/Our approach) from the experiment on the library system feature model. The average
number of decisions made to configure valid products by Depth-First Traversal is 44.4 (33.2 by our approach), and the ratio of average configuration effort saved by our approach is 21.7% compared to Depth-First Traversal.

<table>
<thead>
<tr>
<th>Group</th>
<th>Average Decisions</th>
<th>Average Decisions reduced(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-First</td>
<td>42.4</td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td>33.2</td>
<td>21.7%</td>
</tr>
</tbody>
</table>

4.5.2 Random Systems

We have also conducted an experiment on randomly generated feature models. The random feature model is generated in three steps. First a random graph is generated, where we start with a fixed set of vertices and add edges to the graph based on an edge-probability parameter. Here, a vertex represents a variable feature, while an edge represents a dependency relationship in a graph. In the second step, we randomly generated the relationships between the features. There are four types of relationship that we have to consider here: 1) requires, 2) excludes, 3) parent-children relationship, 4) multiplicity of a variation point. Of course, it is easy to see that here, it is very likely that a large number of conflicts will be generated, so we have to run through the third step, in which we go through all the cycles in the generated graphs to check for inconsistency. In a directed graph, a cycle possibly represents a conflict. Therefore, we have to remove a randomly selected edge per cycle, to destroy the cycle, thus removing the conflict. Once we have completed these three steps, we then have a valid feature model for configuration.

The configuration process is automated as well. A computer program randomly selects a feature and then checks to see if the selection conflicts with previous decisions. If yes, then the program will make a different selection, or randomly change the previous decisions to remove the conflicts. The program repeats the process until all the features are visited.
When using our proposed approach, there will be no rollbacks.

Obviously, the final products generated by the random configuration can be quite different. i.e. one product could contain a large number of features, while the others might contain significantly fewer features. To maintain the similarity of the final products, we reject those products (i.e. invalid products) which contain more than 85% of features or less than 50% of the features of the feature model.

We generated three random systems of 800, 2000 and 3000 nodes, with different edge probabilities. We then configured 1000 valid products, using each of the configuration methods on the three random systems. We included the average numbers of the decisions of our approach/Depth-First Traversal in Table 4.7. The first column indicates the edge probability when creating the random graph, and the corresponding cells represent the average decision numbers of both approaches (i.e. our approach vs Depth-First). With an increasing edge probability, we can see that the decisions made between two approaches become closer. This is because, as the number of edges is increasing, the dependency relationships among the variation points become denser, thus the random selection is more likely to attain large coverage, and the gaps decrease. From Table 4.7, it is quite clear that using our approach is more efficient in the product configuration.

<table>
<thead>
<tr>
<th>Edge Probability</th>
<th>Random graph with 800 nodes</th>
<th>Random graph with 2000 nodes</th>
<th>Random graph with 3000 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>264/413</td>
<td>330/473</td>
<td>361/498</td>
</tr>
<tr>
<td>0.02</td>
<td>219/307</td>
<td>255/374</td>
<td>298/411</td>
</tr>
<tr>
<td>0.1</td>
<td>124/169</td>
<td>145/186</td>
<td>167/199</td>
</tr>
<tr>
<td>0.2</td>
<td>91/112</td>
<td>122/157</td>
<td>133/145</td>
</tr>
</tbody>
</table>

Table 4.7: Experiment Result of Random Systems
4.6 Summary

4.6.1 Effectiveness of Our Approach

By applying our approach to product configuration, instead of visiting each VP of a feature model, we only go through a set of VPs that the algorithm has chosen. In this way, the number of decisions to be made is reduced, saving effort spent on decision-making. Furthermore, this approach is less likely to cause dependency conflicts in the configuration, as we have already incorporated the dependency relationships among the variants in our approach.

4.6.2 Future Work

Currently, our approach only includes two simple relationships. Therefore, how to extend our approach to cover other types of relationships is a challenge. In current feature modelling, not all relationships are well defined, for example, the impact relationship. We are proposing an aspect-oriented feature modelling approach which has better-defined relationships. We believe that we could extend our algorithm once the work on the aspect-oriented feature modelling is completed.

As we have explained at the beginning of the chapter, we did not consider the quality of the final product. If, in the feature model, the feature contributions to the quality attributes are better managed, and we also rearrange the variation points based on the contributions to the quality factor, in this case, the variation points selected are not only based on the coverage, but also based on the impact on the quality. Our approach can be developed to simplify the configuration process, further to rule out the configuration selections which cannot satisfy the NFRs. However, this is totally reliant on completion of the work on modelling the impact of features on quality attributes, especially the impact
of feature groups on quality attributes.

Conducting a real-world experiment is also an important task to be completed in the near future. As we have mentioned previously, we are developing a product line for City Evolution Project, and a case study will be conducted once the product line is developed.
Chapter 5

Quality-Oriented SPL

Architecture

5.1 Introduction

SPL is expected to be a systematic software reuse approach. Two main driving forces behind SPL architecture development are reusability and modifiability (Matinlazzi et al., 2002). It is essential to understand how reusable the reference architecture of an SPL is, related to potential changes, and also to ensure that NFRs of an SPL have been fully considered and addressed in the SPLA design process.

To improve quality-oriented SPL development, the definition of how to achieve related NFRs should be well preserved and transformed between different levels during the whole software development process. For example, how to ensure that the qualified feature configuration will be properly realised and mapped onto the software architecture. There are several aspects that need to be considered. Firstly, the traceability between the functional features and the corresponding components. Secondly, quality attributes also have their impact on the components at the architectural level. As reference architec-
ture provides a solid basis for member product derivation in SPLE, so a quality-oriented reference architecture design approach is desired.

Quality issues on the architectural level are more complex, as NFRs and quality attributes have a great impact on software architecture. They may affect the way in which the architecture is composed, and the style that is adopted. They have further impacts on component interactions through their connectors and protocols. On the other hand, the architecture itself also has an impact on the quality of the final product. So the impact of quality-related requirements on the components needs to be represented clearly, particularly in a reference architecture design. In most cases, there are complex constraints upon quality attributes, which restrict and affect the achievement of NFRs in the architecture. So we set a purpose of developing a quality-oriented reference architecture to manage these issues within domain engineering.

5.2 Literature Review

There are many approaches that have been proposed for SPL architecture development. Some of them are reviewed here, with a brief introduction to each one.

5.2.1 FORM

Feature-Oriented Reuse Method (FORM) (Kang et al., 1998) is a well-known systematic method proposed to analyse features in an application domain or an SPL, and to develop domain architectures and components further for reuse. FORM is an extension to FODA (Kang et al., 1990) which includes application engineering for product customisation and derivation.

FORM consists of domain engineering and application engineering. As a feature-oriented approach, domain engineering starts with domain analysis to explore commonal-
ity and variability within a particular domain. Based on the results of domain analysis, reusable assets such as feature model, reference architecture and components will be produced as the output of domain engineering. Another two design phases will follow during application engineering for development of architecture. Features will be specified in four layers; each layer represents a level of a software development hierarchy. In architectural modelling, reference architectures are defined from three levels of abstractions (subsystem model, process model and module model), according to a four-level feature hierarchy. The subsystem model defines the overall system structure by packaging service features into subsystems in a distributed environment. Then, each subsystem is decomposed into a set of processes that can be considered as the features of the operating environment. In the module model, reusable components are created with specifications to define how to integrate them into the application. The architectures of member products are defined by appropriate feature selections, followed by inclusion of suitable software components, based on concerns and requirements from stakeholders.

5.2.2 PuLSE

PuLSE (Bayer et al., 1999) is another well-known methodology which contains multiple subtasks of SPL construction. The subtask for architecture development is known as PuLSE-DSSA, which is basically a scenario-based approach to support a domain-specific software architecture. In PuLSE-DSSA, scenarios are categorised as generic scenarios to represent functional requirements, or property-related scenarios to describe quality aspects. Each generic scenario is augmented with a set of property-related scenarios and ranked with regard to the architectural importance. The architecture development initially starts with a set of generic scenarios. In each following iteration, more generic scenarios will be selected and added to the current architecture candidate, according to
their importance ranking. The architecture candidate will be further refined and extended until the reference architecture supports all generic scenarios. The application of generic scenarios may result in more than one architecture, so the property-related scenarios which augment them will be considered to evaluate and rank these architecture candidates. The result of the ranking would be an architecture candidate (or a group of candidates), i.e. the reference architecture, ready for further evolution.

5.2.3 KobrA

KobrA (Atkinson and Muthig, 2002) is a component-based approach to modelling software architectures. It is a simple and practical method, which can be used for both single-system and software product families, by applying standard UML models and some commercial tools to enhance its applicability.

Each component, known as a Komponent in KobrA, is described at two levels of abstraction: the specification level, which defines Komponents behaviours and services, and the realisation level, which describes how to fulfill the upper-level services via lower-level Komponents. KobrA has two main engineering activities: framework engineering activities and application engineering activities. A framework engineering activity creates a general framework, which includes all the product variants of a product family. In framework engineering, the specification of a Komponent is described by a set of models. Variabilities are determined by a decision model, whether variabilities can be captured by the existing decisions or it is necessary to add new decisions to the decision model. With the support of a set of models such as an interaction model, structural model, activity model, and decision model, it is able to specify how to design a single Komponent. The purpose of the application engineering activity is to implement the framework in order to derive member products from the family. UML-based graphical models are generated as
the result of both engineering activities, to describe components and also variabilities.

5.2.4 Model-driven Approach

Braganca and Machado (2007) proposed a model-driven and use case-based approach which extends the Four Step Rule Set (4SRS) method to provide a transformation from system requirements to software architecture and design elements. System requirements are modelled with use case and activity diagrams to identify the relationships between use cases. In the analysis, use cases are transformed into use-case realisations, and each use-case realisation represents a set of analysis classes that perform use-case behaviour. Use-case realisations represent the transition from the problem domain to the solution domain. They are considered as the only input for architecture design. Then the logical architecture is developed by integrating various use-case realisations. Following that, analysis elements in the use-case realisation are transformed into specific design elements. Moreover, a feature model is also produced at the analysis stage, based on requirements and use cases. Use cases are used as middle-ware between system requirements and architecture, this preserves the traceability of an SPL.

5.2.5 Pattern-based Approach

Hallsteinsen et al. (2004) also proposed an architectural approach which uses architectural patterns as building blocks for product family architecture design, modelling and use. The relationships between patterns and quality attributes are also considered to support the representation and specification of the reference architecture of product families, to ensure that NFRs will be achieved by the architecture. Architecture patterns always contain reusable solutions for specific design issues, so they are expected to manage recurring problems in software product families. In this approach, the term “pattern” is limited to
the context of product family, so that the architects have a better idea of how these patterns affect the quality properties of derived products. To manage variability of architecture, patterns are categorised as base patterns and varying patterns. Similar to mandatory features, base patterns must be used by all applications. Varying patterns are further considered as optional patterns and alternative patterns. It is interesting that alternative patterns are a set of patterns that basically solve the same problem, but in different ways with different quality parameters. So developers are able to choose the pattern that best fits the application requirements. Scenarios are used to link quality attributes and patterns in the design guideline, as one scenario may relate to several quality attributes and may also relate to patterns in the pattern languages. In the architecture, patterns define roles and collaboration between roles, and roles are normally filled by component implementations. So architectural components are specified by the synthesis of a set of roles from different patterns, and possibly also by collaboration models related to the functionality of the system.

5.2.6 Summary

These SPL architecture methods and approaches are designed from different perspectives, such as feature-oriented, quality-driven, or component-based views. Some methods only describe a general process, while others provide more specific details of engineering activities and steps. The similarity is that all these methods contain multiple views and abstractions for reference architecture development and member product derivation.

Each method has its strength. On the other hand, each of them also contains limitations. For example, FORM provides a whole framework from domain engineering to the design phases. However, it is not specific enough to include the complete process or any extra tool support. KobrA also lacks required process details. Moreover, as a component-
based approach, considerations on the requirements level are not covered. A common issue is that none of these approaches keeps quality as the driver through the design process.

5.3 QADA

Quality-driven Architecture Design and quality Analysis (QADA) (Matinlassi et al., 2002) is a quality-oriented SPL method for both architecture design and quality analysis. According to the authors, software architecture is able to address quality issues of software, therefore, it must be developed and documented properly. As a quality-driven method, architectural styles and patterns will be utilised as the main considerations for high-quality architecture development.

QADA improves SPLA development on two abstraction levels, i.e. conceptual level and concrete level, in order to create high architectural descriptions and analyse architectural quality. As our interest is mainly in the reference architecture development, the investigation is concentrated on the conceptual architecture design and analysis.

QADA employs the concept of view-based software architecture design (Krutchen, 1995; Jaaksi et al., 1999; Hofmeister et al., 2000) to describe software architectures and to separate design concerns. Architectural descriptions are produced at both abstraction levels through three views: a structural view, a behaviour view and a deployment view. The structural view is mainly dealing with software components and system composition. The behaviour view considers the dynamic actions of systems, and components’ responsibilities and behaviours patterns. The deployment view refers to allocation of software components to various deployment units and the computing environment. Each view produces corresponding architectural descriptions which compose the whole documentation of software architecture. Quality attributes are categorised and considered within a related view. Some quality attributes must be satisfied based on specific requirements from cus-
tomers, e.g. adaptability and reusability, while others may be achieved by the interactions and responsibilities of components, e.g. performance and reliability of systems.

QADA starts from requirements engineering to capture the technical properties (functional and non-functional requirements) and the scope of the product line. The technical properties define what functional/non-functional goals the future systems will provide. The technical requirements need to be analysed to establish the impact of each requirement on the architecture and the dependencies between requirements, especially the non-functional ones. By “requirement dependencies”, is meant that system requirements often depend on each other. In order to achieve a particular one, a system may need to fulfill some other requirements. System constraints, such as standards, rules and quality attributes, should also be identified in requirement engineering as they may restrict the design of software architecture. The scopes of the product lines need to be defined to decide if they are suitable for different technologies and businesses. To analyse product line scopes, a customer value analysis (CVA) is conducted to seek customers’ needs with product design, and to realise further the difference between designers’ assumptions of customer value and the customers’ actual perceptions. The relative benefit of inclusion or exclusion of features also needs to be evaluated as a part of scope identification for product lines.

Coming down to the conceptual architectural level, conceptual software architecture is designed to map functionality and quality responsibilities to conceptual structural components (the structural view), collaboration between components (the behaviour view) and allocation of components to hardware (the deployment view). The goal of the structural view on the conceptual level is to describe software component composition and component relationships. The system structure is described as a hierarchy of a decomposition model, which is generated by identifying functional responsibilities and architectural styles
to match certain NFRs respectively. The behaviour view on the conceptual level, specifies the dynamic actions of systems, the behaviour pattern that the systems produce, and ordered sequence of system actions. All this data is represented in a collaboration model. To create a collaboration model, collaboration scenarios need to be identified to represent the most essential sequences of systems actions. Each scenario will be transformed into a collaboration diagram, to represent the scenario graphically. The deployment view is used to group conceptual components into units of deployment, identify the distribution of hardware nodes in the system and specify allocation of deployment units in computing units. Here, each deployment node is a platform for various services and a deployment unit is composed of one or more conceptual components. An allocation model is used to represent a table of units of deployment and to describe the allowed allocation of units. The output of conceptual architecture design is a design rationale, which contains a set of design principles and rules for the product family, to explain why a certain standard has been selected or to describe the selected architectural styles with their preference.

In QADA, the quality analysis of the conceptual architecture contains three sub-tasks: variability analysis, architecture analysis and architectural styles and patterns analysis. In SPLA development, variation points on the reference architecture need to be represented in order to prepare for configuration. The main concerns that variability analysis deals with include what could vary in the architecture and how it could vary, where variation could occur, and when to make variants selections. Architecture analysis is conducted by a quality attribute-based method which evaluates software architecture, based on the relationships between quality attributes and software architecture. Architectural concerns are firstly identified to address the related quality attributes and NFRs of SPLA. Attribute-specific factors need to be linked to represent the impact of architectural styles and patterns on the identified concerns. The objective of architectural styles and patterns analysis is
to build a knowledge base for architectural styles and patterns evaluation in terms of
goodness factors and concerns, and anticipation of their use. The knowledge base is able
to make architecture design more predictable, to have a standard set of attribute-based
analysis questions, and to tighten the link between design and analysis. To improve the
reusability of conceptual architecture, the knowledge base should include architecture-
specific experience to provide a set of packages with solutions to deal with commonly
recurring design issues.

QADA method provides an explicit and quality-driven link between software require-
ments and software architecture. It is important to identify potential risks of reference
architecture refinement for configuration, and to verify that NFRs have been properly
addressed during the reference architecture development. So some extra views are ex-
pected, based on particular SPL domains and related quality attributes, to capture all
business and design data relevant to quality-oriented architecture development. We have
proposed an extension based on this, to represent more quality-related information on an
architectural level.

5.4 Enhancements of QADA

To improve quality-oriented SPLA design, we enhance the work of the QADA method by
introducing two additional views: a quality view and a tradeoff view (Tan et al., 2012).
The proposed quality view provides requirement traceability in the reference architecture
and helps member product configuration. The tradeoff view deals with quality attribute
tradeoff issues in SPL. A tradeoff decision could cause big changes in system architecture,
and thus has to be managed properly.
5.4.1 Quality View

The purpose of the quality view is to provide an overall picture of system structural packages, and to illustrate how they fulfill the quality requirements of member products. In other words, the impact of quality requirements on the reference architecture at the conceptual level is specified in the quality view. The quality view contains many sub-views. For each quality attribute, the related parts of the reference architecture are included in the sub-view. For example, the configurable components, ports and connections which are related to the given quality attribute are included in the sub-view. The sub-view also contains configuration options corresponding to the different ways that components interact with each other to satisfy the quality requirements at different levels. The quality view establishes requirements traceability, especially between the non-functional requirements and architectural design options.

Fig. 5.2 shows how we develop the quality view. We start from requirement engineering. The critical quality requirements, so-called architectural drivers, should be identified. Architectural drivers are usually a combination of those requirements that have a strong impact on the architecture. As some requirements have more influence on the architecture than others, it is crucial to start the architecture design from these architectural drivers.
The architecture drivers need to be embedded into the reference architecture, and later be refined for member product architecture development. The general quality scenarios are used for identifying the reference architecture drivers. A quality scenario is a quality requirement of the associated quality attribute. The quality scenarios can be categorised into general scenarios and concrete scenarios. General scenarios are system-independent, and can be applied to all member products in a product family. Concrete scenarios are derived from the general scenarios. They are application-specific instances and used as quality requirements for a specific member system. To develop the reference architecture, we will focus mainly on general scenarios.

Once we have all the architecture drivers, then the mechanisms will be used to map the architectural drivers to the reference architecture. Like the design patterns, a mechanism contains a solution or possibly multiple solutions to a given problem. For example, a strategy-oriented mechanism describes design strategies to realise the architecture drivers. A concrete mechanism contains the actual components, connection types and their responsibilities for satisfying an architecture driver. From these mechanisms, we can identify the reusable components, connections and the common properties of the product line. Components which contribute to the related architecture drivers, can be grouped. These groups are the subsystem options. The reference architecture of a product line is composed by various subsystem and design options. To manage variabilities, we shall allocate related
scenarios to the components. Thus the variations of component can be identified, which leads to further clarification of the interactions between the components. It is clear that, in the process, we maintain the traceability between the quality requirements and the design of the architecture. The quality view explicitly represents this traceability, which is useful for the software engineers to understand and use the architecture.

5.4.2 Tradeoff View

To obtain qualified architecture options, it is necessary to manage the issues about quality attributes tradeoff in SPLs. Dropping one quality level in return for enhancing other quality standards are tradeoff issues. The intended goal of the tradeoff view is to evaluate the quality attribute relationships, in order to support the member product architecture development. The impact of changes in quality requirements on system architectures needs to be analysed, in order to facilitate software product derivation. The proposed tradeoff view includes the information of quality-related decisions and functional selections.

![Figure 5.3: Developing Tradeoff View](image)

Fig. 5.3 shows the steps in developing the tradeoff view. Quality scenarios identified from the quality view can be used as the input to the quality tradeoff process. Quality scenarios are prioritised by both users and domain experts to explore the importance of each scenario. The intention of scenario prioritisation is to provide a general picture of how to handle the conflicts between quality requirements. If a conflict exists between
two scenarios, the requirements of higher-priority scenario should be satisfied rather than those of the lower priority ones.

Quality-related design decisions affect corresponding components, which have a further impact on the structure of the system. Moreover, relationships between quality attributes could be quite complex. For example, high reusability normally leads to weaker reliability, but enhances the flexibility and testability of systems. Table 5.1 is an extension from the work by Zulzalil et al. (2008), which represents the constraint relationships between quality attributes. “+” indicates that two attributes have a positive impact, “-” indicates they have a negative impact, and “o” indicates they are independent of each other, which does not generate any conflicts. Table 5.1 is helpful in understanding the tradeoff options. Note that only general one-to-one relationships are included. The relationships among multiple quality attributes should be studied more carefully, based on desired system requirements.

| Functionality | Reliability Usability Efficiency Maintainability Portability Testability Flexibility |
|---------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
|               | +                               | +                               | -                                | +                                | o                                | +                                | o                                |
| Reliability   | +                               | o                               | +                                | o                                | +                                | +                                |                                 |
| Usability     | -                               | o                               | o                                | -                                | o                                | o                                | o                                |
| Efficiency    |                                 |                                 |                                   |                                   |                                   |                                   |                                   |
| Maintainability|                                 |                                 |                                   |                                   |                                   |                                   |                                   |
| Portability   |                                 |                                 |                                   |                                   |                                   |                                   |                                   |
| Testability   |                                 |                                 |                                   |                                   |                                   |                                   |                                   |

When deriving member product architectures based on the specific requirements, the reference architecture has to be refined and configured. However, very often, vague requirements are given, and several possible architectural options are available. This happens even when relatively accurate requirements are available. These architectural options, together with their impact on the rest of the architecture, need to be presented explicitly, to help the design of the member product architectures, and also possibly for requirements elicitation. The constraints between quality attributes can be identified from the corresponding scenarios. Valid scenario combinations refer to a collection of corresponding
architecture drivers, and then link to the appropriate mechanisms, which in turn, lead to the design of a valid architectural option. Conflict scenario combinations reveal the conflict requirements, which indicate the tradeoff issues. These tradeoff options generate the architecture recommendations. The tradeoff view consists of multiple sub-views. For each set of related quality attributes, the sub-view contains the corresponding architecture recommendations. In the architecture, the system components and the connections are grouped to match the functionalities of the software system. The components that affect two or more conflicting quality attributes are called tradeoff points (Kazman et al., 2000). Tradeoff points represent critical decisions in the architecture. For instance, we could have a component implement a backup database. It is a tradeoff point in the architecture, because the component affects the reliability positively, but at the same time, affects performance of the system negatively. At each tradeoff point, a set of architecture recommendations is given. The recommendations summarise valid sub-system alternatives that achieve both system functionalities and quality requirements.

Table 5.2: Architecture Options by Tradeoff

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>A1 = 7</td>
<td>A2 = 5</td>
<td>A3 = 6</td>
</tr>
<tr>
<td>Security</td>
<td>B1 = 2</td>
<td>B2 = 3</td>
<td>B3 = 4</td>
</tr>
<tr>
<td></td>
<td>A2 &lt; A3 &lt; A1, B1 &lt; B2 &lt; B3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 presents an example of architecture alternatives. Assuming that the expected system has scalability ranging from 6-7 and the security level is from 2-5, obviously only S1 and S3 meet this target. The architectural recommendations can then be obtained from the tradeoff view. These refine the reference architecture for product derivation.
5.5 Case Study

In this section, we use an example to illustrate how our approach works. Picture Archiving and Communication System (PACS) is a medical imaging system which performs various tasks in medical applications. A typical PACS contains subsystems such as image acquisition, storage, and display, which are integrated by digital networks. PACS is a quality-sensitive system, so that associated quality attributes have to be considered in system architecture design. Because of the complexity of the whole system, we only include partial quality-related variabilities and components of the system to demonstrate our approach.

The goal of the quality view is to clarify the relationships between those components and quality attributes. Fig. 5.5 illustrates the reference architecture with subsystems and components. By matching scenarios and quality factors with system components, the relationships between components and quality factors to be included in the quality view are identified.

From Table 5.3, we can see some of the quality attributes, and the system components and connectors contributing to the realisation of the quality attributes. For example, the Expert System component has a big impact on the system’s usability, and the connection protocols which realise this responsibility could be VPN (Virtual Private Network) or SSL.
Figure 5.5: The Partial Reference Architecture of PACS

(Secure Socket Layer). The Expert System is able to improve the efficiency of systems by providing pre-scanning and prioritisation, in addition to improving the accuracy of the system by providing reliable diagnosis recommendations. Moreover, the Expert System has an impact on the systems’ security considerations, because of issues such as data encryption, network security and authentication. This information is all included in the quality view, so it is clear to the developer that the architectural subsystem relates to the quality attributes of the software product line.

<table>
<thead>
<tr>
<th>Component</th>
<th>Connector</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>Image Storage</td>
<td>Transmission</td>
</tr>
<tr>
<td></td>
<td>Expert System</td>
<td>Web-based communication</td>
</tr>
<tr>
<td>Performance</td>
<td>Image Process</td>
<td>Image scan</td>
</tr>
<tr>
<td></td>
<td>Data Backup</td>
<td>Data Distribution</td>
</tr>
<tr>
<td>Security</td>
<td>Network</td>
<td>Communication</td>
</tr>
<tr>
<td></td>
<td>Security</td>
<td>Encryption</td>
</tr>
<tr>
<td>Cost</td>
<td>Image Storage</td>
<td>Transmission</td>
</tr>
</tbody>
</table>

Table 5.3: Quality Attributes and Related Components (connectors, ports)

To develop the tradeoff view, we start with quality scenarios. Suppose that we are given two scenarios (S1, S2) with different quality considerations.

**S1:** To protect each patient’s privacy, image data has to be encrypted before being stored in the data base. Only authorised requests are accepted to access data from storage,
with approved authentication.

**S2:** To improve the efficiency of system performance, an internal request such as Expert System, is able to access raw data from image storage before data encryption.

From these scenarios, we need to identify the architecture driver, which is the information impacting the architecture design. For example, in our case, we could identify the needs of encryption, authentication, data backup and data remote access. For each architecture driver, mechanisms give out the recommended solutions. For instance, for encryption, options such as public key encryption and private key encryption will be given by the corresponding mechanisms. Each solution recommended by a mechanism has a positive impact on some quality attributes and a negative impact on others. For example, image reconstructors are responsible for calculating 3D images by using a sequence of 2D images. There are two options available: a real-time reconstructor and a portable reconstructor. The Real-time reconstructor is faster, and the portable reconstructor is slower, but supports a wide range of third-party components and applications. So here we have to consider the tradeoff between the system performance and flexibility of the system. In the tradeoff view, we need to present a variety of tradeoff options and the impact of each option on the rest of the system.

The conflicts between quality attributes could be quite complicated. We use the tradeoff between “Performance” and “Security” to illustrate how the tradeoffs impact on the reference architecture. Fig. 5.6 represents two architectural options, which have different “Performance” and “Security” levels. Option1 implements a higher security level, all image data having to be encrypted immediately before storage. So the encryption of the “Security” component is involved in the first place. The “Image Process” component will require the encrypted data passed through the “Security” component by decryption. The
encrypted data will then be transferred to “Image Storage” subsystem for data archiving. The “Expert System” subsystem will not interact with “Image Storage” directly, but all data access requests will go through the “Security” component. The system’s performance will be affected negatively, since there are intervening processing steps and a longer data transmission time before diagnosis. Option2 is an alternative solution, whereby images acquired will be stored directly as raw data. The “Image Process” and the “Expert System” are able to access these data before they are encrypted. “Image Process” only needs to get processing authority through the authentication of “Security”. Obviously the system’s performance is improved in Option2. Patients are able to get the result of the diagnosis much faster than with Option1, but the system’s security level is reduced. This is the sort of information useful in determining the quality of the final product and making tradeoff decisions. Of course, each recommended option also has an impact on other quality attributes of the software system and the interaction among the components could be quite complicated.

![Figure 5.6: Security and Performance Tradeoff](image)

In the real-world, full design of the PACS product line, tradeoff situations are more complicated, as more quality attributes have to be considered, and even minor quality differences will change or compromise the system components and their connections. By listing the quality-related sub-views and combinations, the architectural recommendations are clarified and available to the member product architecture design at the concrete level.
5.6 Aspect-Oriented Architecture Design Approaches

To improve the current SPLA development, some aspect-oriented approaches have been introduced. Aspect-oriented programming enhances the modularity of the object-oriented programming, by explicit modelling of factors crosscutting multiple modules in the program. Similarly, aspect-oriented architecture design attempts to model the factors which might crosscut multiple components of software systems. Aspect-oriented approaches first identify system requirements which might have complicated relationships with components, then design a set of components which can be integrated into the structure of the system to satisfy the requirements identified initially. The integration is normally done by inserting the components through the pointcuts, which are specially selected places in the architecture. If we have a set of possible pointcuts defined, the set components can be plugged into the architecture at various pointcuts. The combination gives us a way to satisfy different requirements or a requirement at different levels. This is very useful to SPLE where we have to manage minor differences between the products. Next, we will review some of the aspect-oriented approaches to architecture design, both for a single system and for SPLE.

5.6.1 Aspect-Oriented Architecture Approaches for Non-SPL Systems

Aspect-oriented architecture approaches provide advanced structures to improve modularity and to separate design concerns. ASAAM (Tekinerdogan, 2004) is built on SAAM (Kazman et al., 1996) to identify architectural concerns early in the software life-cycle. ASAAM is a scenario-based architecture analysis method, which manages architectural aspects explicitly. ASAAM takes a problem description, an architecture description and a set of requirement statements as input to an analysis phase. The aim is to determine the quality factors and potential aspects affecting the architecture design. In ASAAM,
one of the main sources of design is from scenarios. Scenarios from various stakeholders are collected, which represent both actual uses and anticipated uses of the software architecture. A set of heuristic rules is provided to classify the scenarios into direct scenarios, indirect scenarios and aspectual scenarios. Sometimes we could even derive the architectural aspects directly from this step. Aspectual scenarios are derived from direct or indirect scenarios, and represent potential aspects. The scenario interactions also need to be assessed, to identify whether the architecture supports an appropriate separation of concerns, including both non-crosscutting concerns and architectural concerns (aspects).

Once all scenarios are classified, ASAAM focuses on the interactions between scenarios and components, to categorise components based on the heuristic rules defined. Examples are: cohesive component, tangled component, composite component, or ill-defined component. Note that one component could be categorised in different roles in different scenarios, based on these interactions. Since all the architectural aspects and related components are defined, a refactoring process is conducted to represent aspects explicitly. This could be done by using conventional abstraction techniques, such as design patterns or architectural styles.

The Perspectival Concern-Space (PCS) framework (Kande, 2003) is a concern-oriented approach which applies the multi-dimension separation of concerns technique to modelling architecture transformations, in various views. In the PCS framework, each perspective represents software concerns from a specific viewpoint, and the software architecture design is composed in terms of a multidimensional space of concerns from different stakeholders. Each stakeholder holds his own concerns for the system. All concerns are matched to different perspectives and documented in a viewpoint schema document. The main task of the viewpoint schema is to describe and analyse the relevant concerns from different sources. Critical issues need to be highlighted by the current viewpoint on the system.
By studying this document, it is possible to identify motivations for decision-making, address architectural issues, and specify relationships between concerns. With the integration of multiple viewpoint schemas, the concerns from different perspectives and models are matched. This is an important step to understanding the impact of the concerns on the architecture and the cross-cutting nature of the concerns. Eventually, through the mapping of the concerns among different perspectives, a sufficient composition of concerns will be assembled, and lead to simplified perspectival representations. The PCS framework will greatly help the designer to understand the most complicated factors in the architecture design and assist in trade-off issues in the design.

Barais et al. (2004) proposed an aspect-oriented framework, known as TranSAT, for evolution of software architecture to incorporate new concerns and architecture transformation. This framework refines software architecture specifications by inter-weaving technical concerns (NFRS) and provides context independent integration rules. The framework contains several key elements to enhance the evolving definition of software architectures. A software architecture model is built to define software components with their structural interfaces and external behaviour specifications. This hierarchical component model defines software structures in terms of prototypes and communication messages between components. To deal with the difficulties in architecture evolution, aspect-oriented principles are used to identify where a new concern should be coupled to a software architecture. The point-cut definition identifies the interaction space between components and concerns. A point-cut label is associated with a point-cut mask that identifies a set of structural, behavioural, and architectural constraints on a software architecture model. All the integration rules are defined in an asset called an adapter, which contains architecture transformation rules in its configuration interface. To specify software evolution, architect needs to select a concern with a related adapter, then a point-cut label on the architecture
needs to be defined, and necessary information for architecture transformation needs to be provided to the adapter. Finally, point-cut definitions are linked to point-cut labels by the architect, which represents the inter-weaving of integration rules and target software architecture. Software evolution may contain several iterative steps; each step is associated with a specific weaver, dealing with a specific technical integration concern. The order of weaving needs to be figured out by the architect. Each evolution step defines a new component-based software architecture.

There are some other aspect-oriented architecture approaches, such as AODM (Stein et al., 2002), AOCE (Grundy, 2000), and AOSDUC (Jacobson and Ng, 2005). For more details, refer to the comprehensive survey by Chitchyan et al. (2005).

5.6.2 Aspect-Oriented SPLA Approaches

Luckily, there are also some aspect-oriented approaches proposed for SPLA development. Feature models summarise all the requirements for the product line, thus, most of the aspect-oriented architecture design approaches take feature models as their information sources.

Kulesza et al. (2007) proposed a model-based generative approach to map features to aspects and to support automatic software architecture derivation for product families. This approach contains a set of modularisation guidelines to implement the framework in SPL, and also provides mapping rules between features and implementation elements. The framework is structured as a composition of a core structure and a set of extensions. The core structure contains a set of classes crosscut by aspects. A set of extension join points (EJPs) can be used to extend functionalities by establishing a contract between the framework classes and aspects. The use of EJPs is able to increase the integration of the framework with other artefacts, such as components and frameworks. The composi-
tion between the core and the extensions is accomplished by different types of extension aspects and each one defines a crosscutting composition through its exposed EJPs. This framework provides the mapping rules between framework elements and feature models. For example, a framework core and aspects in the core are mapped as mandatory features in a feature model, while EJPs are mapped as joinpoint features. This framework also supports derivation of member product architecture by customising architecture variabilities. The first step in this approach is to specify an architecture model which is used to align the implementation elements with the specification of its architectural components. In the architecture model, the implementation elements are linked to feature models, and the component variabilities are expressed. The second step is to specify the feature model for the AO software family architecture. This approach extends the feature model by defining two properties of features (<<crosscutting>>and <<joinpoint>>) to enable the aspect customisation for product derivation. A crosscutting feature represents an aspect from the architecture model that can extend the behaviour of other system features. A joinpoint feature represents specific join points from implementation elements of the software family architecture. These join points are candidates for extension by aspects in the solution space. A configuration model is then used to specify the mapping between features and implementation elements, based on a set of mapping rules. The configuration model contains a set of elements, such as dependency relationships between implementation elements and features; crosscutting relationships between joinpoint and crosscutting features; and the specification of the mapping between joinpoint features and concrete join points.

Aspect-Oriented Generative Approach (AOGA) (Kulesza et al., 2004) is an architecture-centric approach that has been implemented in multi-agent systems to support modularisation of concerns from early development stages. The multi-agent systems can be seen
as a specific domain so the concept of the approach is general and can be studied for general AOSD. The main consideration of the approach is to improve the domain modelling and architecture specification of crosscutting features from early development phases. The development of this generative approach is able to support the modelling and implementation of both non-crosscutting and crosscutting features uniformly. In the problem space, the feature model is extended and described by a new domain-specific language, called agent-DSL, to collect both non-crosscutting and crosscutting features. In the solution space, the software architecture is designed by using aspect-oriented abstractions to model features and using programming languages such as AspectJ to implement components of the architecture. The approach maintains the mapping from abstractions in the agent-DSL to components and aspects of the architecture. This approach consists of three main processes: domain analysis and specification, architecture design, and implementation. The main task of the first process is to specify crosscutting features as aspectual features; the contribution of architecture design is to define the corresponding architectural aspects by means of a UML-based notation; in the implementation, several methodologies are involved, such as code generators, pre-defined frameworks, pre-defined components, and code templates to automate the code generation. For the architecture development, AO-GA provides a UML-based language to specify and describe AO software architectures. The language provides notation and semantics to build an architectural model for the key components of AO systems. In the architectural model, the components are divided into normal components and aspectual components. Aspectual components are actual aspects at the architecture level and each aspectual component crosscuts more than one component on the architecture level. Interfaces of components are also defined for architecture construction. In this approach, interfaces are also categorised into two groups, based on their crosscutting natures: normal interfaces and crosscutting interfaces. The crosscutting
interfaces are displayed as small grey circles and normal interfaces are displayed as small white circles. A normal interface only provides services to other components. Crosscutting interfaces also specify the relationships between aspectual components and normal components, e.g., aspectual components crosscut normal components through crosscutting interfaces. The focus of this approach on the architectural level maintains two purposes: 1) to work out the specification of the central components of the AO systems; 2) to define the interfaces of the architectural components. In the implementation stage, a code generator is used as an Eclipse plugin to map abstractions of the agent-DSL to compositions of components and aspects in the architecture. This plugin is able to read the description XML file of the agent-DSL and generate classes that represent the elements of an XML Schema.

Sanchez et al. (2007) also proposed an aspect-oriented approach for software product families. The focus of this approach relies on the abstractions and mechanisms to better represent variabilities on the architectural level across the member products. This approach introduces a so-called “AMPLE” architecture design language, which includes several metamodels, to specify the software architecture of an SPL. Three metamodels included are a software architecture design metamodel, a variability metamodel and a constraint metamodel. The software architecture design metamodel is basically intended to design the reference architecture for the whole product family, which contains the mechanisms to enable both commonalities and variabilities on the architectural level. In this case, UML 2.0 is selected as the SPLA design language, to allow flexibility. An architectural orthogonal variability metamodel is used to specify the variability of an SPL without modifying software architecture models. To provide solid mechanisms for relating variant points on the architecture, the variability pointcut language (VPL) is proposed and used to represent the realisation of variabilities on the architecture by architectural
decomposition. It establishes the links between a software architecture description and the points of variability specified by a cardinality-based feature model. A constraint metamodel mainly supports the definition of constraints between variability elements, for example, cardinality-based feature models with “requires” and “excludes” constraints. These variability dependencies are expressed inside the VPL with related constructs.

Looking at these architecture design approaches, most of them take feature models directly as the information source. However, we know that the dependencies among the features could be complex and hard to represent. The incompleteness in feature modelling will certainly affect the architecture design. Thus we propose to model explicitly the crosscutting concerns in the feature model, and then transfer these concerns to the architecture level as the main information source for the design.

Next, we will briefly present our framework. There are still many details to investigate, but we believe that our approach which introduces aspect-oriented modelling to both feature modelling and architecture design, gains the following benefits:

- Better and accurate modelling of the requirements. The crosscutting concerns at the requirements level is explicitly represented.
- More specific information available for reference architecture design.
- More straightforward transformation between the crosscutting concerns at requirements level and architectural level.
- Better traceability from the architecture design decisions to the requirements.

5.7 Proposed Framework

We propose an aspect-oriented framework to improve modelling of crosscutting features and NFRs in current domain engineering. In this framework, we also expect to have appro-
appropriate mechanisms to achieve systematic mapping and transformation from requirement
level to architecture level. Compared to other AO approaches, our framework provides
a significant increment in capability. Our approach starts from a feature model on the
requirement level, and provides traceability of NFRs and crosscutting concerns onto the
architectural level. The framework that we are proposing will support systematic SPL
development in domain engineering by linking software requirements to architecture de-
velopment in a comprehensive process.

The proposed framework is composed of two parts: an AO feature model and an
AO reference architecture development. The main expectation of AO feature modelling
is to convert a conventional feature model into three components, e.g. concrete features,
aspectual features and aspectual concerns. We also expect to define the impact of aspectual
concerns on both concrete and aspectual features. The complex feature relationships
will be decomposed into three levels: NFRs to aspectual concerns, aspectual concerns to
cancrete and aspectual features, and also aspectual features to concrete features. The goal
of AO reference architecture is to specify the mappings from the AO feature model onto
reference architecture. The crosscutting concerns at architecture level will be identified and
the impact on the reference architecture will be presented. The options in the reference
architecture for satisfying the NFRs and quality factors at different levels will also be
presented.

5.7.1 Aspect-Oriented Feature Model

To develop AO feature modelling, we start from functional requirement modelling by us-
ing use case diagrams and activity diagrams. As in the feature model, use case modelling
is generated, based on system requirements. It is able to specify the interactions be-
tween the system and the actors. In our approach, we extend the conventional use case
and decompose it into small business flows. We are interested in internal processes and responsibilities of the subsystems. To apply use case models to a product line, variabilities existing in use cases need to be addressed. The internal business processes need to be specified in order to realise the desired behaviours patterns within the system, which makes aspectual feature identification possible. If a particular system behaviour exists in multiple business processes and interacts with many other user-visible functionalities of the system, this would be a candidate for an aspectual feature.

To understand the AO feature model more clearly, we propose to include crosscutting concerns in the feature model and divide features into several categories, as follows:

- **Concrete Feature**: A concrete feature represents basic functionalities of a product family. These features represent the fundamental services which the systems provide. In the AO context, these features are crosscut by concerns.

- **Aspectual Feature**: Aspectual features also represent the functionality of a product line, and these features crosscut other features.

- **Aspectual Concerns**: Aspectual concerns are the key requirements and system considerations crosscutting the concrete features and aspectual features. The aspectual concerns map the NFRs and requirements on qualities to the features in the feature model. The concerns describe the impact of NFRs and quality requirements on the system composition and the way in which features interact with each other.

According to these feature categories, the tasks of AO feature modelling include developing related models to treat crosscutting features (functional) and concerns (non-functional) in a systematic way. The first step is to model concrete features, which is relatively easy, as concrete features correspond to the user-visible functionalities of the system. Current use case modelling and scenarios-based approaches are able to address
the functional requirements of the software systems. The main task in this step is to develop a model which is able to represent clearly the concrete features and relationships between these concrete features.

The second step is to identify and modularise aspectual concerns, which is one of the major contributions of our framework. Aspectual concerns could be identified to describe the impact of NFRs on the functional features in the NFR modelling process. In the framework, the contribution of each concrete feature to satisfying the NFRs will be examined in multiple viewpoints of the system. The reason for introducing a multiple viewpoint approach is that this kind of approach is helpful in managing the inconsistencies and conflicts in the requirements. Since all combined concrete feature options are identified, we are able to specify the related quality levels that can be achieved by these options. Based on the identified relationships between concrete features and NFRs, aspectual concern candidates can be identified. An aspectual concern corresponds to system requirements. It describes the impact of an requirement on concrete features and NFRs, i.e. how these features interact with each other, and the level of quality and NFRs that it satisfies by interacting in the desired way. From the user’s point of view, the aspectual concerns are considerations for system configuration; they are akin to the relationships between the features and NFRs.

The last step of AO feature modelling is to identify aspectual features and to map aspectual concerns to concrete features and aspectual features. An aspectual feature is a feature that crosscuts other concrete features, i.e. it impacts the behaviour of other features. They are additional responsibilities that do not affect the main business flow. The distinction between aspectual features and concrete features is that an aspectual feature has to contribute to related aspectual concerns. There are several situations in the aspectual concerns mapping. An aspectual concern is mapped to an aspectual feature
(aspect-to-aspect), then the aspectual feature is used by concrete features to satisfy certain requirements as the aspectual feature crosscuts these concrete features. An aspectual concern could crosscut several concrete features (aspect-to-feature). This situation suggests several implementation options for concrete features, to satisfy different requirements. Another situation arises when several aspectual concerns crosscut the same concrete feature (aspects-to-feature). Conflicts could exist in this situation and affect the achievement of some functional and non-functional requirements.

By the end of the process, we will have converted the feature model into three components, i.e. concrete features, aspectual features and aspectual concerns. The impact of aspectual concerns on both types of features will be identified as well. Moreover, the complex feature relationships will be decomposed into relationships among these three components and NFRs. Following our aspectual feature modelling, NFRs and quality attributes are better understood and better modelled, which directly improves the quality of product configuration. Furthermore, the traceability between features and requirements (functional and non-functional) is preserved, which leads to further AO reference architecture development.

5.7.2 Aspect-Oriented Reference Architecture

An aspect-oriented reference architecture framework specifies how features and aspectual concerns from AO feature modelling are mapped onto a reference architecture. The key point is to identify crosscutting concerns caused at architecture level, and how reference architecture is affected and composed accordingly.

Concrete features identified from the AO feature model need to be mapped onto the components at the architectural level, which means that the functionalities represented by the features will be implemented by related components. There are plenty of conventional
architecture design approaches able to design the components and provide the mappings. Aspectual concern mapping is more difficult in terms of reflecting NFRs and aspectual feature transformation.

To transfer the aspectual features in an AO feature model onto the reference architecture, we mainly rely on the aspectual concerns identified in the feature model. All the aspectual concerns need to be addressed at the architecture level. Similarly to architecture drivers of a single system, these aspectual concerns are the architectural factors and considerations that have a great impact on the architecture of systems. Meanwhile, they also have an impact on other architectural factors and components, i.e. crosscutting. All these impacts are reflected in the reference architecture by having corresponding pointcuts on architectural components. The pointcut represents a point at which the control flow could be interrupted and all constraints on the crosscutting could be identified. Having the component pointcut on the architecture and the options of various components being plugged in at the pointcut, the reference architecture describes all the possible architectures that can be derived. Each of these architectures will lead to a software system with its own quality attributes.

To map the aspectual concerns on to the architecture level, we will examine how the system reacts to a user’s input, to satisfy functional and non-functional requirements at different levels. We look at the impact of aspectual concerns on the components and detail the collaborations of components in terms of pointcuts and possible components to be plugged in. The aspectual features form the additional task to complete in addition to the main workflow. Thus, these sorts of features become components which crosscut other components in most cases, e.g. they are the ones to be inserted at the pointcut.

Here we only describe the basic concept of how to derive an AO reference architecture from an AO feature model. The first task of the framework is to model functional
requirements by developing components and subsystems to implement concrete features. Conventional SPL approaches can be used to transform concrete features into components, or subsystems to meet functional requirements. The output of this task will be components and connectors to link these components. This step will provide the base for the rest of the development.

The next step is to develop a set of architecture scenarios to describe systems. From the use case and business processes identified, we are able to work out a set of scenarios describing the internal behaviours of a system. Since the requirements are for multiple products, it is common that we have multiple alternative flows for a use case corresponding to the different requirements and quality expectations. These alternative flows of the business process suggest alternative internal workflows of the system, which will be mapped to parallel scenarios.

The following step is to identify variabilities of components, and possibly pointcuts, on the architecture components. Components of the architectural model from the first step are involved in scenarios, so it is possible to specify the responsibilities of these components and their interactions. One of the benefits of using parallel scenarios is that the variations of a component and its interfaces can be identified by examining parallel scenarios. Some of the variability can be located within a single component, i.e. there are different implementations of the component or different possible ways of interacting with the rest of the system. Some of the variabilities cannot be managed this way, since they suggest extra components and pointcuts in the system. Here we need to examine how the aspectual concerns are addressed in the architecture. These aspectual concerns suggest ways in which the components interact, and possible ways in which the components can crosscut each other to satisfy functional and non-functional requirements at different levels. So, these crosscutting concerns will be used to identify more components and to
refine the components’ interactions.

The final step is to assess scenario interactions, to refine potential, larger scope cross-cutting concerns. Scenarios could be combined to implement more complex functionalities, so examining the interactions between scenarios could address some complex aspectual concerns. After this step, all the identified concerns should be properly addressed in the reference architecture, in terms of component crosscutting relationships. Clearly, all the designs can be traced back to the feature model, and the traceability from the feature model to the reference architecture is preserved. We also suggest an architecture-refactoring process at the end, to tidy up and to improve the quality of the whole architecture.

5.8 Summary

Quality-based reference architecture development is receiving more attention, as it is able to realise stakeholders’ expectations, and to guarantee the quality of software products. To achieve this goal, we extend the QADA method, by introducing quality-based design methods as extra views. The benefit of this design method is to maintain the requirement traceability and to improve member product derivation. This task still has the following steps to work on, as we would like to investigate how to incorporate the variabilities of product line into the two additional views that we have proposed. Furthermore, we would like to investigate how to develop member product architectures from the reference architecture.

We believe that the aspect-oriented approach is a suitable method for SPLA design. However, to achieve better reference architecture, and to have the quality modelled more explicitly, we suggest to extend the aspect-oriented modelling to feature modelling, which allows us to better model the requirements and variability among the member products. Our suggested framework improves the requirement traceability, furthermore, as the im-
pact of non-functional requirements and functional features is explicitly managed in the feature model as aspects, we believe that our proposed approach manages the quality of the final product more effectively.
Conclusion and Future Work

6.1 Conclusion

Software Product Line Engineering (SPLE) is developed as a means to improve software development, and to increase software quality by systematic software reuse. Due to the existence of variations in software product lines, it is more difficult to efficiently evaluate the quality of member products. To preserve software quality in the development process, quality issues should be managed as early as possible in SPLE, and the relationships between software assets and quality attributes need to be addressed and represented precisely. Domain engineering of SPLE is considered suitable for dealing with quality issues and providing a well-defined foundation for software development.

This research focuses mainly on managing the relationships and impacts between quality attributes and software assets in domain engineering. To provide a comprehensive quality-related framework, quality attributes need to be firstly addressed on the requirement level, then inherited and mapped on the architectural level of SPLs. This thesis contributes to current SPLE development in several ways.

- **Quality-oriented Feature Evaluation.** We have proposed an approach of link-
ing quality attributes to features and measuring features’ contributions to quality and NFRs realisations. This approach specifies features’ relative importance to corresponding quality attributes in terms of a complete ranking list for each quality attribute. These features’ rankings are able to help product configuration to satisfy various quality attributes, and also to monitor quality tradeoffs. We have compared our approach with other approaches, and the result shows that our approach is able to handle quality evaluation more effectively by incorporating information from various sources.

- **Efficient Product Configuration.** The overall quality of feature-based configuration is preserved by minimising potential errors and conflicts in the configuration process. Fixing configuration errors may cause propagation of side-effects, especially in real-world, large feature models. We proposed an approach to minimise this kind of error by taking feature dependencies and constraints into account, to identify a small set of key features. By making decisions on these key features, the rest of the features will be automatically decided, based on their relationships with key features. Our approach saves configuration effort by improving the efficiency of product configuration. The results of our experiments were positive in terms of an enhanced configuration process, even in real large random systems.

- **Quality-oriented Reference Architecture Development.** The reference architecture of an SPL provides a solid foundation for customisation of member products. Once we have determined the relationships between quality attributes and components, it will be much easier to achieve the expected software quality during the process of product derivation. To achieve this, we have proposed a quality-oriented reference architecture approach, based on an existing quality-driven architecture
method - QADA (Matinlassi et al., 2002). Our approach contains two views: a quality view and a tradeoff view. The quality view starts with architectural drivers, which are some critical quality requirements that have a great impact on the architecture. Related components and components interactions need to be identified by associated mechanisms, in order to achieve these quality requirements. The quality view contains multiple subsystems, which comprise the overall reference architecture. The tradeoff view deals with quality tradeoff issues, by investigating the relationships between quality attributes. The tradeoff view produces architecture recommendations for different levels of quality attribute achievements.

6.2 Future Work

In the current research, we have mainly considered the quality-related issues in feature modelling and reference architecture design of SPLE. Some of the contributions have been summarised above. However, there are still many open problems and unsolved issues. For example, currently, I have been dealing with simple feature relationships mostly, e.g. “requires” and “excludes”. The more complicated relationships are not included in my work. Moreover, we have not been able to specify the details of the impact of features on the quality attributes. To model the quality attributes in domain modelling, we have proposed a framework in Chapter 5, which uses aspect-orient techniques to model the crosscutting relationships. This will firstly bring an extra layer in the domain modelling, which will deal with more complex relationships between the features, and also between feature groups and quality attributes. Features can be grouped and their contributions to quality attributes can be evaluated as a whole, especially in the case of some crosscutting relationships between features and quality attributes. Our proposed framework also supports the reference architecture design better as the aspects at requirement level will be
explicitly transformed to the architectural level. Additionally, variability in an SPL will be better handled, as NFRs and quality attributes will be used as a basis for representation of various forms of software structures.

To develop such an aspect-oriented framework, there are some main tasks need to be accomplished, which are listed below and followed by some of the key research questions to be answered.

- **Aspectual Feature Modelling**
  1. Concrete feature modelling.
  3. Mappings from aspectual concerns to both functional and aspectual features.
  4. Features and aspects interdependency representation.

- **Aspectual Reference Architecture**
  1. Functional requirement modelling and architecture development.
  2. Scenarios development and architectural aspect identification.
  3. Scenarios evaluation and component refinement.
  5. Variability representation in aspect-oriented architecture.

- **Framework Integration**
  1. Conversion from aspectual feature model to aspectual reference architecture.
  2. Related rules to transform and map aspects identified from requirement level to architectural level.
  3. Necessary mechanisms to ensure the completeness and accuracy of variability representation from feature model to reference architecture.
Research Questions:

1. how to identify mapping from aspectual concerns onto both concrete features and aspectual features?

2. how to represent NFRs and quality attributes in cases where aspectual features crosscut concrete features?

3. how to identify and represent crosscutting concerns variabilities at architectural level?

4. how to clarify transformation between AO feature modelling and AO architecture?

5. how to represent aspect, cutpoints, joinpoints etc. in reference architecture?

Once we have answers to these research questions, we believe that this aspect-oriented framework is able to support the quality modeling more efficiently in domain engineering. Some of the preliminary work has been accepted by the Conference of ASWEC 2013. We hope that we can complete this work in a short time.
Bibliography


Tan, L., Lin, Y. Q. and Ye, H. L. (2012), Modeling Quality Attributes in Software Product Line Architecture, in ‘2012 Spring World Congress on Engineering and Technology (SCET2012)’, Xi’an, China.


Zhang, G. H., Ye, H. L. and Lin, Y. Q. (2010), Quality Attributes Assessment for Feature-
Based Product Configuration in Software Product Line, in ‘the 2010 Asia Pacific Software Engineering Conference’.