COMPONENT-BASED REFERENCE ARCHITECTURE TOOL FOR SOFTWARE PRODUCT LINE ENGINEERING

A dissertation submitted to The University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences

2013

MERT BALCI

School of Computer Science
Table of Contents

Abstract 6
Declaration 7
Intellectual Property Statement 8
Acknowledgement 9
1. Introduction 10
  1.1 Project Aims and Objectives 11
  1.2 Report Outline 12
2. Software Product Line Engineering 13
  2.1 General Principles of Software Product Lines 13
  2.2 Traditional Software Reuse Vs. Software Product Lines 15
  2.3 Software Product Line Organization 16
  2.4 Software Product Line Requirements Engineering 19
  2.5 Variability Management 21
    2.5.1 Categories of Variability 21
    2.5.2 Levels of Variability 22
    2.5.3 Mechanism of Variability 23
  2.6 Software Product Line Feature Model 24
  2.7 Software Product Line Functional Model 25
  2.8 Software Product Line Reference Architecture 26
  2.9 Software Product Line Component Development and System Integration 28
  2.10 Software Product Line Costs and Benefits 29
  2.11 Current Software Product Line Approaches 31
    2.11.1 FAST 31
    2.11.2 FODA 32
    2.11.3 RSEB 33
    2.11.4 FeatuRSEB 34
    2.11.5 ConIPF 34
    2.11.6 PuLSE 35
    2.11.7 KobrA 36
    2.11.8 Evaluation of current approaches 37
  2.12 Summary 37
3. Component-based Approach to Software Product Line Engineering 39
  3.1 The Main Approach 39
List Of Figures

2.1 The two-life-cycle model of software product line engineering 14
2.2 Essential Product Line activities 15
2.3 Asset responsibility in the development department model 17
2.4 Asset responsibility in the business unit model 17
2.5 Asset responsibility in the domain engineering unit model 18
2.6 Asset responsibility in the hierarchical domain engineering unit model 19
2.7 An example of the Feature Model 24
2.8 Data Context Diagram 25
2.9 Control Flow Diagram 26
2.10 Three basic techniques to capture variation points 28
2.11 Activities and deliverables in component development 29
2.12 Economics of software product line engineering 30
2.13 FAST Process flow 31
2.14 The functional model with the fundamental elements 33
2.15 The main phases of PuLSE 36
3.1 A suggested X-MAN component-based approach for domain engineering 41
3.2 X-MAN components and connectors 42
3.3 Data Transformation in FODA and X-MAN 43
3.4 An example of a data transformation in FODA and X-MAN 44
3.5 Sequencer connector in X-MAN and related state transition diagram 45
3.6 pure::variants screenshot which depicts a feature model 46
3.7 X-MAN Reference Architecture Tool 47
4.1 X-MAN RAT main window when it first runs 50
4.2 A part of the main window of X-MAN RAT 51
4.3 Feature model shown in the tree control 51
4.4 Possible products based on feature model 52
4.5 Different views of X-MAN RAT 52
4.6 Design Item Creator 53
4.7 An example of a feature model in X-MAN RAT 54
4.8 Feature and link types 55
4.9 A part of the main window when reference architecture is selected 57
4.10 Reference architecture design item window 57
4.11 Data Input-Output Window for data channels 58
4.12 An example of a reference architecture in X-MAN RAT 58
4.13 Compiler window for reference architecture items 59
4.14 An example of a product feature tree in X-MAN RAT 60
4.15 An example of a product architecture in X-MAN RAT 61
5.1 Home Automation Systems feature model in pure::variants 65
5.2 Export wizard in pure::variants 66
5.3 Feature Model of Home Automation System in XML format 67
5.4 X-MAN functional model for Home Automation System 69
5.5 The feature model of Home Automation System in X-MAN RAT 71
5.6 Link types and colours in X-MAN RAT Reference Architecture 74
5.7 X-MAN RAT Reference Architecture for Home Automation System 75
5.8 Selected features of Product_1 77
5.9 Product architecture of Product_1 in X-MAN RAT 78
5.10 Selected features of Product_12 78
5.11 Product architecture of Product_12 in X-MAN RAT 79
5.12 Selected features of Product_30 79
5.13 Product architecture of Product_30 in X-MAN RAT 80
List of Tables

2.1 Various mechanisms suggested by Jacobson 23
5.1 Components with inputs and outputs 68
5.2 Mapping table for components, connectors and features 73
Abstract

Software Product Line Development is a development methodology that focuses on high-level reuse of large-grained software pieces. The organizations can gain many benefits by applying this approach mainly in terms of time, cost, and complexity. Nevertheless, software product line approach has some issues that have to be solved [1]. The main issue is reference architecture that could not appropriately be constructed by following any approach so far. Component-Based Development (CBD) is another software development methodology which builds a system by bringing pre-built components together. One of the strongest aspects of CBD is reusability. CBD can be combined with software product line approach to construct reference architecture. This project is bringing these approaches together in order to suggest an efficient way to software product line development. The scope of the project focuses on creating reference architecture based on X-MAN functional model [43] and the feature models which are developed in pure::variants [46] variability management tool. In order to create reference architecture, a new tool which is called as X-MAN Reference Architecture Tool (X-MAN RAT) will be designed and implemented. The integration of the tools will be achieved by XML files to simplify the creation of the reference architecture. X-MAN RAT will not only construct component-based reference architecture but also generate product specific architectures based on reference architecture.
Declaration

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

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

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

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

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

Firstly, I would like to thank my supervisor, Dr. Kung-Kiu Lau for providing the necessary guidance in completing my dissertation. I am sincerely grateful for all his help throughout the entire project. I believe that the project would not have been successful without his assistance. Particularly, the workshops boosted my knowledge in Software Product Line Engineering.

Next, I would like to thank PHD students from Component-based Software Development Research Group for their great support. They have contributed a lot to the success of this project.

Further, I would like to express my sincere gratitude towards Gökhan Büyükyumukoğlu and Burhan Özgün. Friendship knows no distance and I want to thank them back home for their endless support and providing motivation and great mental strength.

I would like to express my gratitude towards my parents Ali Rıza and Emine, my sisters Günseli and Aslı and my brother Mustafa for their unconditional love and support. I am truly indebted and thankful to them for in taking time to patiently listen to my occasional issues, during the course of this Master’s degree. They have always reminded me to keep an open mind and have assured me of positive results. I feel the luckiest person in the world to have such a great family.

Finally, I would like to thank the Great Almighty for granting me this opportunity and in providing me with great strength, confidence, and positive attitude to fulfill my thesis.
Chapter 1 – Introduction

The software engineering researches have brought many new techniques and methodologies to the software world. While these cutting edge methods are being used for the sake of human being and making the life easier, the complexity in software is increasing dramatically at the same time. From the software development point of view, reusability is one of the most crucial concerns. In addition to that, cost and time to market parameters are of great importance for the industry as well. For that reason, component-based and product line approaches have been adapted to the software development in order to improve the development process, exploit reusability and decrease cost and time as well as business goals [29].

The main idea of Component-based software development (CBSD) is creating a system by bringing smaller parts together in a manageable way in order to reduce complexity. These parts are called as components which have complete functionality and developed independently from each other. Software reuse is another important aspect of CBSD. The components are mainly developed to be reusable. Therefore, not only increased productivity and quality but also decreased time and cost are achieved [33, 48, 49].

Software product line engineering (SPLE) focuses on high-level software reuse in the aim of improving development costs and time to market by managing commonality and variability among the products of the product line [2]. A successful SPLE brings many benefits by helping organizations achieving large-scale productivity gains, improving time to market, maintaining market presence, nourishing extraordinary growth, increasing product quality, improving customer satisfaction, achieving reuse goals and enabling mass customization [1]. SPLE is based on the two-life-cycle model. The first life cycle is called Domain Engineering (development for reuse). The second life cycle is called Application Engineering (development with reuse). The domain engineering process is based on exploiting the commonality and variability between the products in the product line as well as creating reference architecture which reflects these common and different features explicitly. The application engineering process focuses on deriving particular products by using the analysis and the reference architecture of the domain engineering phase. There are a
number of Software Product Line approaches. FODA [22] is the most common approach among these approaches. However, there is hardly any method which describes all the phases in details. The link between features, functional model and reference architecture has remained weak in all of the methodologies. Moreover, there is no executable architecture which can be created directly [1, 2, 33].

This project will try to find a method to combine component-based approach and Software Product Line Engineering. X-MAN [43] has been selected as the component model and pure::variants [46] variability management tool has been selected for domain engineering phase. The requirements will lead to the feature models which will be created in pure::variants [46]. Functional model will be created in X-MAN component model. The reference architecture will be created by using these feature and functional models in a new tool which is called as X-MAN Reference Architecture Tool (X-MAN RAT). X-MAN RAT will be designed and implemented to achieve the main goal. Furthermore, the new tool will create product specific architecture based on reference architecture.

1.1 Project Aims and Objectives

The main goal of the project is to investigate different software product line development approaches and propose a method which focuses on Component-based Software Product Line Engineering. In order to accomplish the goals in the limited time, the ultimate goal is to create reference architecture based on feature models and X-MAN [43] functional models. In addition to that, a new tool which is called as X-MAN Reference Architecture Tool (X-MAN RAT) will be created to accomplish the main goal. The objectives to achieve the goals of the project are defined:

1. Investigate the different Software Product Line (SPL) approaches available in the literature.
2. Investigate how to manage variability in these SPL methodologies.
3. Explore X-MAN component-based model.
4. Explore FODA method.
5. Explore pure::variants variability management tool.
6. Propose a new method to create component-based reference architecture.
7. Create a new tool to construct reference architecture (X-MAN RAT).
8. Extend X-MAN RAT to make it support importing the feature models in XML format.
9. Extend X-MAN RAT to make it create product specific architectures based on reference architecture.

1.2 Report Outline

The rest of the report is structured as follows:

- Chapter 2 (Background) explains The Software Product Line concepts and different approaches. It starts with Software Product Lines general principals. Next, different aspects of Software Product Line, namely organization, requirements, feature model, functional model, reference architecture, variability management techniques, costs and benefits are described. Finally, a number of Software Product Line projects and methodologies are presented.
- Chapter 3 (Component-based Approach to SPLE) describes the methodology of the project, X-MAN model, the need for the new tool, project tools and the integration of the tools respectively.
- Chapter 4 (X-MAN Reference Architecture Tool) introduces the new tool which is called X-MAN RAT and how to use it to create reference architecture and product specific architecture. It also details the creation of reference architecture according to a number of defined rules.
- Chapter 5 (Case Study: Home Automation System) walks through the implementation of Home Automation System and explains how to create reference architecture and product specific architectures in X-MAN RAT.
- Chapter 6 (Conclusion) sums up component-based SPLE approach. It is finalised with future work part.
Chapter 2 - Software Product Line Engineering

In this chapter, I try to explain the Software Product Line (SPL) concepts. It starts with Software Product Lines general principals. Next, different aspects of Software Product Line namely organization, requirements, architecture, variability management techniques, costs and benefits are described. Finally, a number of Software Product Line projects and methodologies are presented.

2.1 General Principles of Software Product Lines

Software product line development is a methodology that creates a collection of similar software products by using a set of common assets and a same production technique. Clement et al. (2001) describe Software Product Line as follows, "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [1].

The main purpose of Software Product Line Engineering is large-grained software reuse. It uses domain knowledge of business to exploit commonalities of the products in the product family and to distinguish the differences among these products. In Software Product Line Engineering approach, it is intended to acquire organizational benefits by using commonalities and variations between the products that are offered to a specific market segment. Manufacturing industry has been using product lines since a long time ago to manage cost and productivity by taking advantage of common features between products. Boeing and Ford are the examples of companies that use product line approach. Even though product line is not a new concept, it is relatively new for the software industry. Nonetheless, it has been proved by studies that organizations which apply product line approach can reap great benefits and gain significant improvements in customer satisfaction, development costs, development time, reliability of the product, usability, portability and maintenance [1, 2, 3].

In his major study, Bosch (2000) points out that development of a product line can be broken down into three dimensions which are the fundamental reusable assets of the
product line, the organizational management of the product line and the life cycles of the reusable assets of the product line [3]. According to van der Linden (2007), Software Product Line Engineering is based on the two-life-cycle model [2]. The first life cycle is called Domain Engineering (development for reuse). The second life cycle is called Application Engineering (development with reuse). As shown in figure 2.1, domain engineering consists of five main processes which are product management, domain requirements engineering, domain design, domain realisation and domain testing. Application engineering consists of four main processes which are application requirements engineering, application design, application realisation and application testing. All of the common assets are used to establish the infrastructure of the product line. In the application engineering phase, the final products are built on the top of the product line infrastructure which is created in the domain engineering phase.

![Figure 2.1: The two-life-cycle model of software product line engineering][1]

In her introduction to Software Engineering Institute’s Software Product Line approach, Northrop (2002) describes the three essential activities of Software Product Lines as follows, “At the highest level of generality are three essential and highly iterative activities that blend technology and business practices. Fielding a
product line involves core asset development and product development using the core assets under the aegis of technical and organizational management” [4]. Figure 2.2 illustrates those three essential activities. The core asset development is usually called domain engineering and product development is called application engineering. The activities are more parallel rather than sequential and substantially affect each other. During the life cycle of product line they evolve by that interaction [1, 4].

![Diagram of Essential Product Line activities](image)

Figure 2.2: Essential Product Line activities according to Northrop [4].

2.2 Software Product Lines Vs Traditional Software Reuse

Both traditional software development and software product line can rely on reuse of the assets. However, in traditional software development, reuse is not planned and traced. Even though, it may have reusable algorithms, components, methods used in the development which are stored in a repository, it can be longer to find and adapt a reusable asset from that repository than implementing a new functionality to the system [3]. Contrary to traditional software development, software product lines plan, manage and ensure reuse of assets. Decisions are made deliberately in a systematic way with a strategy. In addition to what has been said, according to the Software Engineering Institute’s Framework 5.0:
“When we speak of software product lines, we don't mean

- fortuitous, small-grained reuse
- single-system development with reuse
- just component-based or service-based development
- just a reconfigurable architecture
- releases and versions of single products
- just a set of technical standards [5]“

The core assets of a product line are the artefacts that are stored in the repository. These reusable assets cost too much to develop from scratch. The examples of these artefacts are architecture, requirements, domain models, components, test cases, test files, performance criteria and project management documents. All of these items have the definition in terms of commonality and variability for different products in the product line. The main idea is to instantiate the product by the artefacts rather than coding large amount of functionality [1, 5].

2.3 Software Product Line Organization

Software reuse is such a critical and important process that it must be planned and managed in a very careful manner. There cannot be any coincidence in software reuse. Therefore, it goes without saying that it requires major changes in the organization [7].

There are different approaches to manage the organization and cope with the difficulties in the software product line approach. Bosch (2000) offers a number of structures for the organizations which apply software product line methodologies in terms of the size of the development department. The models of the structures suggested by Bosch (2000) are the development department model, the business unit model, the domain engineering unit model and the hierarchical domain engineering unit model [3].

The development department model suggests that all the developers can work on both developing domain assets and products. The developers all belong to a single development department. When the number of the developers exceeds 30, some crucial changes in the organization is inevitable. Figure 2.3 illustrates the development department model.
Figure 2.3: Asset responsibility in the development department model according to Bosch [3].

The business unit model, as shown in figure 2.4, suggests separated units of the development team when the team is usually consist of from 30 to 100 developers. Each of the units develops a particular product based on the product line. There is no unit that focuses primarily on the product line platform. However, that can lead to some serious problems. The most probable problem is developing product-oriented assets that are against the main purpose of the software product line engineering. In the long run, product line can lose its reusable assets and face the severe corrosion problem in the core assets.

Figure 2.4: Asset responsibility in the business unit model according to Bosch (2000) [3].
The domain engineering unit model generally fits well to the development teams that are consist of over 100 developers. There are two main groups as shown in figure 2.5. The first group is called domain engineering unit and works solely on the development of core assets. The second group is called business unit which works on the specific products of the product line. When the developers in the domain engineering unit unnecessarily focus on the core assets and overlook the customer needs, it can be a serious problem for the product development as it will be difficult and even impossible to reflect the customer needs to the products.

Figure 2.5: Asset responsibility in the domain engineering unit model according to Bosch [3].

Similar to the domain engineering unit model, the hierarchical domain engineering unit comprises two main groups that are domain engineering unit and business unit as shown in figure 2.6. However, due to the fact that this model is more suitable for the development teams in which there are several hundreds of developers, there can be some specialized domain engineering units when team has to deal with large number of variability and the products. The main issue with this model is that it can be quite complex to manage the units.
Another approach to manage the organization was offered by van der Linden (2007). According to van der Linden, there can be three types of organizations that are product-oriented organization, process-oriented organization and matrix organization. To be successful in the implementation of software product line, the most crucial aspects of the organizational models are assigning the roles and responsibilities in the right way. Another indispensible aspect is the communication in the organization. It must be transparent and managed in an explicit manner [2].

2.4 Software Product Line Requirements Engineering

Requirement Engineering is of great importance for the successful Software Product Lines as it helps to reduce time, complexity and risk problems. Without applying requirement engineering, it is hardly achieved gaining positive results from software product line approach. The worst thing would be the failure of the project. Requirement Engineering ensures software product lines to gather the correct requirements and produce well-defined products in a managed and planned way [1, 8, 9].

There is a general description of a requirement defined by Harvell (1997) as follows, “If it mandates that something must be accomplished, transformed, produced or provided, it is a requirement – period [10]“. From the software systems perspective, requirements are used to model system behaviours [11]. During requirement
engineering phase, the focus is on defining the problem and scope rather than suggesting a way to solve the problem and build the application. It can be assumed that it is a very simple process. Nevertheless, when it is not applied carefully, it can increase the complexity and be a failure reason for the project [12].

Sommerville (1997) describes the scope of software requirements engineering as “all of the activities involved in discovering, documenting and maintaining a set of requirements of a computer-based system. The use of the term ‘engineering’ implies that systematic and repeatable techniques should be used to ensure that system requirements are complete, consistent, relevant, etc [11]”. According to him, even though there is not a widely accepted general requirements engineering process, it has some typical phases namely elicitation, analysis, specification, validation and management [11].

In the Framework for Software Product Line Practice, Version 5.0, Software Engineering Institute suggests that requirements should be seen as the core assets of the product line. That is to say that requirements should be managed separately like the other core assets and are of great importance for reusability. Requirements can have variation points that are set of deltas for different products [5].

Software Engineering Institute describes the phases of requirements engineering that are elicitation, analysis, specification, validation and management as follows:

• Requirements elicitation is the process of discovering, reviewing, documenting, and understanding the user's needs and constraints for the system.
• Requirements analysis is the process of refining the user's needs and constraints.
• Requirements specification is the process of documenting the user's needs and constraints clearly and precisely.
• Requirements verification is the process of ensuring that the system requirements are complete, correct, consistent, and clear.
• Requirements management is the process of scheduling, coordinating, and documenting the requirements engineering activities (that is, elicitation, analysis, specification, and verification) [5].”

In addition to these definitions, the main focus during all these processes should be on exploiting the common features and variations, reusability of the requirements, change management and traceability. That is what is peculiar to the software product line requirements engineering [1, 5].

2.5 Variability Management

Variability lies at the heart of the software product line development process. Variability management enforce defining and showing the variations between the specific products explicitly. It is entirely engaged with all the processes in the development process. Variation points and variations identify the context of variability in the product line [13]. David M. Weiss and Chi Lai describe variability as, “An assumption about how members of a family may differ from each other [14]”. The telecomunication domain can be a good example to simply illustrate. Considering mobile phone as the product in the product line, the features like the number of keys and dimension of the screens are the variabilities.

As products evolve and new features are needed during the time, it becomes harder to manage variability for both specific products and product line infrastructure of the software product lines. That leads to changes in variations as well. This implies that there are two variation aspects in terms of changes. Hallman and Polls define these two variation aspects as “variation in time and in space” [15]. The different releases of the particular products are related to variation in time. Variation in space is related to two products which have variants that have different implementation version.

2.5.1 Categories of Variability

Variability can be categorized by a number of criteria. According to Felix Bachmann and Len Bass [16], variability can be in:

• function
• data
• control flow
• technology
• the objectives of product quality
• product environment

Therefore, these variability points can be used to classify variability. In a few words, variability in function denotes that some products may have a specific function but others not. Variability in data denotes that a specific data structure can be implemented differently in different products. Variability in control flow denotes that a specific set of interaction can be different in different products. Variability in technology is related to the platform (operating system, hardware, programming language environment, user interface) which can be different in terms of functionality. Variability in the objectives of product quality denotes that concerns like performance, modifiability or security can vary in different products. Variability in product environment denotes that the domain of the product may require particular requirements.

2.5.2 Levels of Variability

Different levels of variability can be presented to make the level of abstraction obvious. Bosch and Svahnberg [17] point out these levels as follows:

• Product line level variability
• Product level variability
• Component level variability
• Sub-Component level variability
• Code level variability

Product line level variability deals with how products become different from each other. For instance, it shows the components which are employed by different products or which product specific code (PCS) are employed.

Product level variability deals with the selection of components and architecture for a specific product. On this level, the product architecture is composed of components. PCS is implemented for the specific product variation.
Component level variability deals with how to add and use different implementations of components and how to extend components life time by designing the interfaces of the components when new concrete implementations are added.

Sub-Component level variability deals with adding new features and removing unnecessary parts to get rid of the dead code in a component.

Code level variability is directly related with evolution and variability among the different products.

2.5.3 Mechanism of Variability

In order to manage variability, various mechanisms are suggested by Jacobson in terms of type of variability, the development stage and the right time to apply [18]. Table 2.1 shows these mechanisms.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time of Specialization</th>
<th>Type of Variability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>At class definition time</td>
<td>Specialization is done by modifying or adding to existing definitions.</td>
</tr>
<tr>
<td>Extension</td>
<td>At requirements time</td>
<td>One use of a system can be defined by adding to the definition of another use.</td>
</tr>
<tr>
<td>Uses</td>
<td>At requirements time</td>
<td>One use of a system can be defined by including the functionality of another use.</td>
</tr>
<tr>
<td>Configuration</td>
<td>Previous to runtime</td>
<td>A separate resource, such as file, is used to specialize the component.</td>
</tr>
<tr>
<td>Parameters</td>
<td>At component implementation time</td>
<td>A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition.</td>
</tr>
<tr>
<td>Template instantiation</td>
<td>At component implementation time</td>
<td>A specification type is written in terms of unbound elements that are supplied when actual use is made of the specification.</td>
</tr>
<tr>
<td>Generation</td>
<td>Before or during runtime</td>
<td>A tool that produces definitions from user input.</td>
</tr>
</tbody>
</table>

Table 2.1: Various mechanisms suggested by Jacobson [18].

Table 2.1 can be seen as a quick reference to the mechanisms of variability in the literature [18].

In this project, the focus of variability will mainly be on components and control flows in reference architecture.
2.6 Software Product Line Feature Model

A feature model is the output of the domain modeling phase. From the end users point of view, the feature model is a clear model to understand and cover the facilities of the domain. Essentially, it encompasses the common features and variations of the products in the domain. The features are represented in a hierarchical tree structure. There are four main feature types that demonstrate the relationship between a parent and a child feature, namely mandatory, alternative, or and optional [22]. These features are described as follows:

- **Mandatory**: The features that are the compulsory requirements that must be the part of every specific product.
- **Alternative**: The features that are a set of specialized features and one of them solely must be selected for the product derivation.
- **Or**: The features that from which at least one of them must be selected for the product derivation.
- **Optional**: The features that are the non-obligatory and can be or not be a part of particular product.

Figure 2.7 shows an example for the feature model. It can be seen from this feature model that catalogue, payment and security are the mandatory features. Search is the optional feature. Payment has two or features that are bank transfer and credit card. Security has two alternative features that are high and standard.

![Feature Model Diagram](image)

Figure 2.7: An example of the Feature Model [33]
In addition to these types of features, the feature model has some compositional rules. For instance, “requires” and “mutually exclusive with” are the rules which are not expressed in the tree structure of the model. These rules describe the relationship between the features semantically [22, 30].

2.7 Software Product Line Functional Model

A functional model demonstrates the functional and behavioral characteristics of a system with data and control flows [31]. Functional characteristics identify inputs, outputs, transformation of the data, flow of the data and the logical structure. Behavioral characteristics identify states, states transitions, events, control flow and conditions. There are two modeling techniques used particularly to model these aspects, namely data flow modeling for functional characteristics and finite state machine modeling for behavioral characteristics.

Data and control models that demonstrate functional and behavioral characteristics of a system are described in more details as follows [31]:

- **Data Model**: It is a model that identifies the functionality by three major elements, namely Data Context Diagram (DCD), Data Flow Diagrams (DFDs), and Process Specifications (PSPEC) for the DFDs [31].
  - *Data Context Diagram (DCD)* models the data flow between the system and the external entities which are the terminators. This diagram can be thought as the summary of the flow. An example of the diagram is illustrated in figure 2.8 in which T1 and T2 are the terminators, S is the system and df1, df2 and df3 are the data flows.

![Figure 2.8: Data Context Diagram](image-url)
- **Data Flow Diagram (DFD)** is used to model data flow in DCD with the refinements. It is composed of a number of basic elements, namely data flow, data transformation and data store.

- **Process Specification (PSPEC)** is used to define the specifications of the functionality in details for data transformations.

- **Control Model**: It is the model that shows the control flow and explains the system behaviour. It comprises three major elements which are Control Context Diagram (CCD), Control Flow Diagrams (CFDs) and Control Specifications (CSPECs) respectively.
  - **Control Context Diagram (CCD)** models the control flow between the system and the external entities which are the terminators.
  - **Control Flow Diagram (CFD)** is used to illustrate CCD with the refinements in a richer format. It comprises a number of basic elements, namely control flow, control transformation and control store. Figure 2.9 shows an example of CFD.

  ![Control Flow Diagram](image)
  
  **Figure 2.9: Control Flow Diagram [33]**

- **Control Specification (CSPEC)** is used to define the specifications of the behaviour in details for control transformations.

### 2.8 Software Product Line Reference Architecture

A software system always requires an architecture that constrains the scope of functionality and determines the quality attributes of the system. Requirements that are in the solution space are placed into the architecture [2]. Bass defines the software architecture as, “The software architecture of a program or computing
system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. Externally visible properties are those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on [19]. It is of great importance for the successful software projects. Therefore, the software architects must design the architecture with utmost care as it will be harder to make changes and succeed in gaining product quality because of these changes later in the project [13].

There are four main concerns for the architecture defined by Jazayeri et al. in [20]:

1. Requirements that are significant for the architecture significant and have essential influence on the architecture.
2. Conceptual architecture which defines the architecture’s key concepts without delving into implementation specifications.
3. Structure which captures the decomposition of the application into components and the relationships between these components.
4. Texture which is a set of rules to create the architecture and evolve it over time.

From the software product line perspective, there is no difference in terms of architecture design. However, a product line architecture fulfils the needs of not only one specific product but also all the products in the product line. It is expected to capture and exhibit common features and variations explicitly in an efficient way [2]. According to van der Linden, “The shared architecture in a product line is called the reference architecture. The reference architecture describes a generalised architecture that provides a solution for the range of products in the product line. It contains the variability that is instantiated in product architecting, although not all of it is visible on the architectural level [2]”. The reference architecture is used to create products. It is instantiated and extended in compliance with the variations for the specific product in order to create the product architecture. It is an output of domain engineering process and a common asset for the application engineering process [2]. As feature model and functional model demonstrate variations, commonalities and behaviours, a proper reference architecture is based on these models.
The architect may have to consider a large number of products at the same time while designing reference architecture. The main challenge is to deal with the conflicts between the requirements. Hence, variability management is of great importance for the reference architecture which has to support variability. It implies that reference architecture must cover all the variation points for all the different products. There are three basic techniques to capture variation points in the reference architecture namely adaptation, replacement and extension as illustrated in figure 2.10 [2].

- The Adaptation Technique: A particular component has only one implementation, but more than one interfaces to change the behaviour.
- The Replacement Technique: There are a number of different implementations of the component. One of the suitable implementation is chosen for the architecture in the application engineering.
- The Extension Technique: The architecture is expected to provide interfaces to add new components.

![Diagram showing adaptation, replacement, and extension techniques](image)

Figure 2.10: Three basic techniques to capture variation points [2].

### 2.9 Software Product Line Component Development and Systems Integration

The components in the product line are developed in order to realize the functionality of the requirements for the products with the required variability. According to Bosch, the components in question are characteristically large-grained and similar to...
object-oriented frameworks rather than relatively small classes in object oriented systems [3]. Besides, these components can not only be a member in the core assets repository but also be developed for the specific products [13]. Figure 2.11 illustrates the activities and the deliverables of component development in an organization that uses software product line approach.

![Activities and deliverables in component development](image)

Software Engineering Institute defines software system integration as follows, “Software system integration refers to the practice of combining individually tested software components into an integrated whole. Software is integrated when components are combined into subsystems or when subsystems are combined into products [5]”. Correspondingly, there are two chief software system integration methods namely the waterfall model and the incremental model. The waterfall model ensures that integration is a distinct process during the development. In contrast, the incremental model supports continuous ongoing integration of the components and subsystems. For the reason that most of the complex integration problems are experienced at the end of the development in the waterfall model, the incremental model is generally chosen model for the system integration [5, 13].

### 2.10 Software Product Line Costs and Benefits

There are many reasons lead companies to start using Software Product Line Engineering approach. It supports large-scale reuse, process improvement, reduced costs and time to market and quality improvement. For the reason that it increases the company’s business power, it goes without saying that the main reason is based on economic considerations. Although software product line approach needs more extra
up-front investment in comparison with traditional approaches, reuse approach is more cost-effective by orders of magnitude. Figure 2.12 illustrates the economics of software product line engineering [2]. It demonstrates that software product line engineering needs some up-front investment. However, approximately after 3 systems implementation, dramatic reduction of cost per system is achieved when compared to the single system development.

![Figure 2.12: Economics of software product line engineering [2].](image)

On the other hand, the cost of the maintenance decreases dramatically. As the majority of the components used in the products are proved to be matured and working well, the defect density is expected to be lower and that leads to developing high quality products as well as more reliable and secure systems. Another benefit aspect of software product line is the consistency of the user interface. That is realized by using the same components for the similar user interaction for the products [1, 2].

According to Software Engineering Institute’s Framework for Software Product Line Practice, Version 5.0, the remarkable benefits gained by the organizations are “large-scale productivity gains, decreased time to market, increased product quality, decreased product risk, increased market agility, increased customer satisfaction, more efficient use of human resources, ability to effect mass customization, ability to maintain market presence and ability to sustain unprecedented growth [5]”.
2.11 Current Software Product Line Approaches

In this section, a number of current Software Product Line approaches are given as an overview. These approaches are respectively FAST [21], FODA [22], RSEB [24], FeatuRSEB [25], ConIPF [26], PuLSE [27], KobrA [28, 29]. While some of them have focus centred on Domain Engineering, the others propose a complete software product line approach.

2.11.1 FAST

Family-oriented abstraction, specification and translation (FAST) [21] is a family-oriented production process suggested by David Weiss in the early 1990s. The main idea is to analyze potential software families and build services for productivity of development of the family members.

FAST comprises three main sub-processes that are domain qualification, domain engineering and application engineering as shown in figure 2.13. In the domain qualification sub-process, an economic model is created by cost analysis. In the domain engineering sub-process, the main focus is on commonality analysis, family definition and product line infrastructure as well as reusable core assets. In the application engineering sub-process, family members are produced by using the reusable core assets.

![Figure 2.13: FAST Process flow [21]](image)
2.11.2 FODA

FODA or Feature-Oriented Domain Analysis [22] is a process which defines the domain and develops products of the family in a structured way. It has been proposed and developed by Kang. FODA comprises three core processes which are analyzing of the domain, analyzing the features and modelling the features. Initially, it starts with scoping the domain and spotting the products which will be the members of the product line. That is achieved in the domain analysis. Following that process, commonality and variability analysis is performed to take the advantage of reusability by applying feature analysis. That process results in a list of common functions and variations of the products. Finally, the features are modelled by using that list which comprises the common and varied characteristics.

There are three major phases which guide to the successful results in FODA, which are context analysis, domain modelling and architecture modelling. These phases are described briefly as follows:

- Context analysis is performed to define the domain and create a context model which contains requirements.
- Domain modelling is performed to model the requirements by using the results of commonality and variability analysis. The feature models are the outputs of this phase.
- Architecture modelling is performed to create the reference architecture by using the feature models. The output which is the reference architecture is used to develop the specific products.

The two major models in FODA are feature model and functional model that have been introduced in the previous sections 2.6 and 2.7.

FODA uses state activity charts and state charts to model functional and behavioural aspects correspondingly. These charts are proposed by Structured Analysis and Design Technique (SADT) [32, 33].
Figure 2.14: The functional model with the fundamental elements [34]

Figure 2.14 shows the basic elements which are used to model functionality. These models are data model and control model respectively. As previously explained in the major phases of FODA, initially, context diagrams are created for both data and control models. Data flow diagrams (DFD) and control flow diagrams (CFD) which are alike DFDs are created after the functions are decomposed. Process and control specifications are used to identify and control functionality and behaviour. The interactions between these models are through control prompt and data condition. In figure 2.12, the data flow is shown by solid lines and the control flow is shown by dashed lines.

2.11.3 RSEB

Reuse-Driven Software Engineering Business (RSEB) [24] is proposed to be a reuse methodology to achieve significant business objectives. In addition, it is expected to develop business performance. The main focus is on the use cases. Firstly, the requirements are described for the product line domain with the help of use cases. Secondly, the architecture is designed in details. Finally, the object models which are related to the use case are created [13, 18].

The Unified Modelling Language (UML) [35] is used to explicitly manage variability that is identified in the use cases and object models. RSEB has a number of steps in which an engineering activity takes place, namely requirements engineering, architecture family engineering, component system engineering and application system engineering. The outputs of these engineering processes are as follows [13]:
• Requirements Engineering: variation points and variations that are defined by use cases
• Architecture Family Engineering: a layered architecture
• Component System Engineering: reusable components
• Application System Engineering: products

2.11.4 FeatuRSEB

Featuring Reuse-Driven Software Engineering Business (FeatureRSEB) has been suggested by Griss et al. in 1998 [25]. It is derived by bringing the FODA [22] and RSEB methods together [24]. FeatuRSEB attaches two more processes that are the domain engineering and feature modelling from FODA to start the RSEB process. Even though RSEB manages variability in use cases in an informal way, there is no feature model created during the process. Therefore, FeatuRSEB uses the feature models of FODA. However, these feature models comprise slightly different diagrams which are illustrated in a tree or a network notation. With the help of these new notations, it represents the variation points explicitly [13].

The domain engineering steps can be summarised in a number of steps. According to Griss et al., the first three steps (1-3) belong to domain analysis and the steps from 4 to 7 belong to component engineering. These steps are represented as follows [25]:

• Domain identification and scoping
• Choosing and analyzing the requirements, examples and trends
• Identification, factoring and classifying the feature sets
• Developing the domain model and architecture
• Representing the variability and commonality
• Exploiting the variability and commonality
• Implementing the reusable components and packaging them

2.11.5 ConIPF

Configuration of Industrial Product Families (ConIPF) is a European FP6 project [26]. In his major paper, Eriksson describes ConIPF as, “a project which wants to integrate both the product line approach and the structure-oriented configuration
technologies [13]”. Similar to the other software product line approaches, the main idea is the development with reuse. Moreover, it adapts configuration methodologies that are developed by applying artificial intelligence [13, 26].

2.11.6 PuLSE

The Product Line Software Engineering (PuLSE) method has been designed by Fraunhofer Institute Experimental Software Engineering (IESE) in late 1990s [27]. PuLSE approach states that the focus of software product line should be on the products rather than organizational aspects. According to Knauber et al. [36], not only large organizations but also relatively small companies can benefit from this methodology.

Eriksson describes the overall philosophy of PuLSE as follows [13]: “

- PuLSE provides a complete framework that covers the whole software product line development life cycle, including reuse infrastructure construction, usage, and evolution.
- PuLSE is modular and customizable: It consists of six technical components that can be selected and instantiated in order to satisfy the needs of specific companies.
- PuLSE can be introduced incrementally by augmenting existing software development processes and products with product line specific aspects step by step.”

PuLSE is composed of three major parts which are the deployment phase, the technical components and the support components respectively. These main parts are shown in figure 2.15. Principally, the deployment phases use the technical and support components with the purpose of detailing a particular software product line [13, 27].
2.11.7 KobrA

The KobrA (Komponentbasierte Anwendungsentwicklung) method is a component-based software product line approach that has been developed by Fraunhofer Institute Experimental Software Engineering (IESE) [28]. The novelty in this approach is the combination of reuse in small concept in component-based approaches and reuse in large concept in software product line methodology [13]. Atkinson et al. argue that "the product-line and component-based approaches to software development seem to have complementary strengths. They both represent powerful techniques to support reuse, but essentially at the opposite ends of the granularity spectrum [37]". KobrA approach is not only a software product line methodology but also a single system development approach.

The KobrA method fundamentally has two major activities which are framework engineering and application engineering [28, 29]. These activities are summarised in the following:

- **Framework Engineering**: In this phase, a generic framework is created. The framework defines the common and variable features explicitly. A set of KobrA components are used statically to construct the framework. The outputs of this phase are a number of framework models which are described in text and UML notations. Product line approached is applied in this phase.
Application Engineering: In this phase, component-based approach is applied in order to initialize the generic framework. The components are specified and realized in two levels. In component specification, the externally visible behavioural aspects and properties are defined. In component realization, the decomposition of the components into the subcomponents is described. The ultimate objective is to build products with particular variations in terms of the specific customer needs. Therefore, the outputs are the product models which are described in text and UML notations.

In their technical report, Muthig et al. demonstrate how to apply KobrA approach by developing a library system product line [38].

2.11.8 Evaluation of current approaches

From the approaches represented above, it is obvious that they follow similar processes. The context analysis is the starting point in common. Domain engineering and application engineering follow the context analysis. One of the main concerns is exploiting the commonalities and variations. Nonetheless, the abstraction level is so high for all of the approaches that there is no detailed guideline to apply these approaches [33].

FODA [22] and FeatuRSEB [25] claim to solve the issues predominantly in domain engineering phase whereas FAST [21], FORM [23] and KobrA [28, 29] claim to provide a complete solution for all the phases of software product line engineering. However, indeed, the domain engineering phases have more attention than application engineering phases [33].

In terms of variability, it is evident that some of the approaches use the feature models. The origin of these feature models that are used in FORM [23] and FeatuRSEB [25] is FODA [22]. On the contrary, PuLSE, KobrA [28, 29] use decision models. FAST [21] manages the variability in a text format by applying commonality analysis [33].

2.12 Summary

In this chapter, software product line concepts and different approaches have been explained. It started with Software Product Lines general principals. Next, different
aspects of Software Product Line, namely organization, requirements, feature model, functional model, reference architecture, variability management techniques, costs and benefits have been described. Lastly, a number of Software Product Line projects and methodologies are presented.

According to the background research, it goes without saying that the abstraction level is so high for all of the approaches that there is no detailed guideline to apply these approaches. Moreover, it is obvious that there is no tool which creates reference architecture using both feature modelling and functional modelling.

Chapter 3 introduced the new approach for software product line engineering.
Chapter 3 - Component-Based Approach to Software Product Line Engineering

This chapter covers the main approach, the relation between FODA functional model and X-MAN functional model, the tools selected for feature modelling and functional modelling, the new tool created for reference architecture and product architecture. Finally, the integration between these tools is explained.

3.1 The Main Approach

In order to accomplish a good implementation for component-based software product line engineering, the proposal of this project covers 4 main steps:

- Feature modelling
- Functional modelling
- Reference architecture based on both feature and functional models
- Product creation based on reference architecture

As discussed in chapter 2, FODA [22] method is one of the well-defined approaches and base to a number of different methodologies. Hence, FODA has been selected to take part in the project. There are two major models in FODA which are feature model and functional model. Feature modelling will be carried out by FODA approach. However, a component-based approach is needed to construct component-based reference architecture as functional modelling is not compositional or modular in FODA. For this reason, functional model that demonstrates the functional and behavioural characteristics of a system with data and control flows should be created based on component-based model.

There are three types of component in terms of components and composition mechanisms in current models [45]:

- An object as component: the components are the objects. The composition is performed by direct method calls. The components can call other components provided services directly. Object-oriented programming uses this composition mechanism.
- An architectural unit as component: the components are the architectural units. The composition is performed by connecting the compatible required
and provided ports and indirect message passing. ADLs use this composition mechanism.

- An encapsulated component: the components are encapsulated. The composition is performed by coordination. The components are completely independent after the composition takes place. X-MAN uses this composition mechanism.

It is obvious that exogenous connectors and encapsulated computation and control make the X-MAN the most efficient and valuable model among all models. In terms of component-based software product line approach, X-MAN comes forward with the advantages it provides [33, 45]. Therefore, X-MAN component model will be used to create functional model and reference architecture for software product line. It is believed that it will bring utmost flexibility to the reference architecture as the components and connectors can be added, replaced or removed with no dependency problem and minimal trouble to the system in this model.

A powerful tool which follows FODA methodology is required in order to simplify feature modelling process. For that reason, pure::variants [46] tool has been selected after a survey in that field.

In [33], it is proposed that, “The FODA functional model can be replaced by the X-MAN component model in order to provide the equivalent specification of functional and behavioural aspects of the domain [33]”. That is to say that, each element in X-MAN can stand for a related element in functional model. It is of great importance to show that functional model in X-MAN is the most suitable selection for the project. This relation is explained in details in section 3.3.

Figure 3.1 shows that suggested X-MAN component-based approach for domain engineering.
Figure 3.1: A suggested X-MAN component-based approach for domain engineering [33].

Base on the study in [33], X-MAN component-based approach is described and discussed in details. However, there is not any defined process for the creation of the reference architecture in this study. By taking the idea and some parts of the suggested methodology in [33], this project tries to prove that the creation of reference architecture is possible and efficient in a new tool which is called X-MAN Reference Architecture Tool (X-MAN RAT). One of the main goals of this project is developing that new tool which is planned to be implemented by using C#.NET [50] and Visual Studio 2012 IDE [51]. According to the steps which are shown in figure 3.1, context analysis and feature modelling are accomplished in pure::variants tool, functional modelling is accomplished in X-MAN GME tool [53], and X-MAN architecture modelling is accomplished in X-MAN RAT in this project. The integration between pure::variants and X-MAN RAT is established by XML file transfer.

3.2 X-MAN Component Model

X-MAN is a hierarchical component model in which components are encapsulated and used to create the model. Data, control and computation are encapsulated by the components [39, 40]. The two major concepts are the encapsulated components and the composition connectors. There are two types of components that are atomic
component and composite component. Composite component is created by using composition connectors [41].

Atomic component is the basic component which is composed by an invocation connector (IC) and a computation unit (CU). Figure 3.2a illustrates the structure of atomic component. The CU has the complete behaviour and functionality which is implemented in a programming language. Therefore, it does not require another component to perform its methods which is why the computation is encapsulated. The invocation connector (IC) is the only way to communicate with the component. The IC provides the methods to the interface of the component. Thus, the methods become executable [41].

![Figure 3.2](image)

(a) Atomic Component  (b) Composite Component

Figure 3.2: X-MAN components and connectors [33].

The composition connector (CC) is used to create composite components. In figure 3.2b, a composite component is depicted. X and Y are the atomic components. IX and IY are the invocation connectors of X and Y. The composition connecter is CC. The composition connector structures the control. It can also be used with the other connectors together in a composite component. An atomic component of a composite component never calls the other atomic component. The CC handles the coordination of execution and encapsulates control. Hence, they are called exogenous connectors. They can be used in the model in a hierarchy [41, 43]. X-MAN model splits the
control and computation in different parts in a component. However, control and data flows are not separated [42].

The X-MAN model has two major composition connectors, namely sequencer and selector. The sequencer offers sequencing and the selector provides branching. In addition to that there are adaptors which are specialized connectors used for adapting a single component. Loop and guard are the examples for the adaptors. In the following, there is a list of the connectors used in X-MAN model [33, 44].

- The invocation connector
- The sequencer composition connector
- The pipe composition connector
- The selector composition connector
- The iterator adaptor connector
- The guard adaptor connector

3.3 FODA Functional Model and X-MAN Component Model

FODA functional model and X-MAN component model can be related to each other in terms of modelling elements. That is to say that, a functional model in FODA can be represented in X-MAN components model. Therefore, when the feature model is formed, the functional model will be created in X-MAN component-model thanks to that convenience. In this section, the examples of the relevant elements are explained with figures.

Data Transformation and Data Flow

Data transformations are used for different functions of the system for transforming inputs to outputs [22].

![Data transformation in FODA and X-MAN](image)

(a) Data transformation in FODA  (b) X-MAN component

Figure 3.3: Data Transformation in FODA and X-MAN [33]
As shown in figure 3.3a, pdt is an example of a data transformation, Ia is the input data and Ob is the output data of that transformation. Equivalently, figure 3.3b represents the data transformation with X-MAN component (CU), Ia is the input data and Ob is the output data of that transformation.

Data flows connect data transformations by getting the output from the first one and passing it to the second one as an input [22].

![Data Flow Diagram](image)

(a) A data flow between two data transformations

![Data Channel Diagram](image)

(b) A data channel between two X-MAN components

Figure 3.4: An example of a data transformation in FODA and X-MAN [33]

As shown in figure 3.4a, c is the data flow and it connects the data transformations, X and Y. Similarly, figure 3.4b shows a data channel that connects two components, X and Y.

**Control Transformations and Signals**

A control transformation is responsible for coordinating control and triggering the data transformations with respect to control signals [22]. State transition diagrams are used for modelling the behaviour of a control transformation. Data transformations are represented as components and behaviours are represented as the control connectors in X-MAN component model [22, 33]. Sequencer connector can be given as an example.
As shown in figure 3.5, there is a sequencer. The actions that range from A1 to An are performed to change the states that range from S1 to Sm. Sequencer starts with A1, performs the rest of the actions in order and finishes with An. During that execution, states change from S1 to S2, S2 to S3, and so on. The last state is Sm.

All the control connectors in X-MAN can be modelled in that way to represent the functional model in X-MAN [33].

3.4 The Need for a New Tool

Based on the literature review in chapter 2, it is obvious that there is no tool which creates reference architecture using both feature modelling and functional modelling. FODA methodology mainly focuses on the domain engineering part. However, there is no compositional or modular functional modelling approach. X-MAN component model can be used to create compositional functional model. Nevertheless, X-MAN component model cannot represent reference architecture although it can show functional model clearly. To solve these problems and create a reference architecture based on both feature models and functional models, a new tool needs to be built.

The name of the new tool is X-MAN Reference Architecture Tool (X-MAN RAT). X-MAN RAT takes a feature model which is created in pure::variants, helps the user create reference architecture and visualize the specific product architectures. It also calculates and list possible products based on the feature model. The details of the
tool are explained in chapter 4. The case study which is implemented by using this tool is explained chapter 5.

3.5 Project Tools

In this section, the tools that are planned to be used to accomplish the goals of the project are presented.

3.5.1 pure::variants

pure::variants is a tool that is used to manage variability. It is defined in [46] as, “pure::variants provides a set of integrated tools to support each phase of the software product-line development process. pure::variants has also been designed as an open framework that integrates with other tools and types of data such as requirements management systems, object-oriented modelling tools, configuration management systems, bug tracking systems, code generators, compilers, UML or SDL descriptions, documentation, source code, etc. [46]”. Figure 3.6 shows a screenshot from pure::variants tool [46].

![Figure 3.6: pure::variants screenshot which depicts a feature model [46].](image)

The feature modelling and the family modelling are the main modelling techniques that are used practice software product line engineering in pure::variants tool. The basic idea is to manage variability explicitly with the feature models. It promotes the
creation of hierarchically structured multiple feature models. A variable architecture can be designed and product variants can be derived. However, this architecture is not in an executable form [46].

As pure::variants solves most of the conflicts during the feature modelling process, it is of great help for the domain engineering part in software product line engineering. The feature models can be transformed into XML files. Therefore, this tool has been used in all of the phases till the reference architecture is created.

3.5.2 X-MAN Reference Architecture Tool

X-MAN Reference Architecture Tool (X-MAN RAT) is a new tool which has been created mainly for the creation of component-based reference architecture based on both feature models and functional models. Figure 3.7 shows the main form of the tool. In chapter 4, the tool is explained in details.

![Figure 3.7: X-MAN Reference Architecture Tool](image)

3.5.3 Integration of pure::variant and X-MAN RAT

The integration has been achieved by XML [47] files which are created in pure::variants. X-MAN RAT can take an XML files, parse it and show the feature model in a tree control. In addition to that, X-MAN RAT calculates the number of the possible products and creates product architecture based on that feature model.
3.6 Summary

In this chapter, the main approach, the relation between FODA functional model and X-MAN functional model, the tools selected for feature modelling and functional modelling, the new tool created for reference architecture and product architecture have been explained. Lastly, the integration between these tools is explained. In chapter 4, the new tool which is X-MAN Reference Architecture Tool is introduced.
Chapter 4 - X-MAN Reference Architecture Tool

In this chapter, X-MAN Reference Architecture Tool (X-MAN RAT) which is one of the main parts of the project is explained. It starts with an introduction to the tool. Next, different functions of the tool for SPLE are presented. As the project chiefly focuses on SPL reference architecture, creating component-based reference architecture is the most important section.

4.1 Introduction to X-MAN RAT

X-MAN Reference Architecture Tool (X-MAN RAT) is a new tool which has been built mainly for the creation of component-based reference architecture based on both feature models and functional models. X-MAN RAT has a number of main elements and functions. Firstly, it can import a feature model, create a list of possible products according to feature types and visualize feature model in feature model view. Secondly, it can create design elements which are X-MAN components and connectors and construct reference architecture by mapping these elements to features in feature model. All of the X-MAN components and connectors in reference architecture exist in functional model which is created in X-MAN component model. In fact, creation of reference architecture in X-MAN RAT starts with adding design items that are X-MAN components and connectors according to functional model. That is to say that, functional model which has to be constructed in X-MAN component model before starting creation of reference architecture is created again in X-MAN RAT. However, these design items are all mapped to the features in the feature model when they are added one by one. Therefore, reference architecture will look like functional model, but it will not be just a functional model. Reference architecture will represent commonalities and variations on connectors and components with different colours visually. It is desired reference architecture as it shows commonalities and variations as well as it demonstrates the functional and behavioural characteristics of a system with data and control flows. Lastly, X-MAN RAT can create product specific architectures when a product is selected from the list. The selected product has a number of certain features. Thus, when it comes to creating the product, reference architecture is used and unrelated components and connectors are removed from the architecture according to the feature of products. All these main elements and functions are explained in the next sections.
As discussed in the previous sections, this tool is designed to create reference architecture and product specific architecture by following a number of steps which are as follows:

- Step 1: Create feature model in pure::variants
- Step 2: Create functional model in X-MAN component model
- Step 3: Import feature model to X-MAN RAT
- Step 4: Visualise feature model in X-MAN RAT
- Step 5: Create reference architecture in X-MAN RAT
- Step 6: Select a specific product to create product specific architecture

All these steps will be explained with a case study in chapter 5 to make the usage of the tool and the creation of reference architecture clear. However, it is important to explain the main elements and the usage of the tool before the case study.

Step 1 and step 2 are accomplished out of the X-MAN RAT. Step 3 is the starting point for the tool. Step 3, 4, 5 and 6 can be accomplished by X-MAN RAT.

![Figure 4.1: X-MAN RAT main window when it first runs](image)

In figure 4.1, X-MAN RAT main window when it first runs is shown.
4.2 Importing Feature Models

The first process is importing the feature model which has been created in pure::variants in XML format. Hence, “Import Feature Model” is pressed to start importing. In figure 4.2, this button can be seen.

![Figure 4.2: A part of the main window of X-MAN RAT](image1)

After pressing the import button, file dialog window is shown and the related XML file of the feature model is selected from the directory. When the selection is done, the application parses the XML file and creates a tree view on the form. Figure 4.3 shows the feature model after import process. The feature model can have different types of features namely mandatory, optional, alternative and or. These features are shown in different colours to have a clear view of the feature model.

![Figure 4.3: Feature model shown in the tree control](image2)
4.3 Possible Products of Software Product Line

The import process also calculates the number of possible products and creates product feature trees for each possible product based on the feature model.

![Possible products based on feature model](image)

**Figure 4.4:** Possible products based on feature model

As shown in figure 4.4, when the import process is done, the tool shows the number of possible products in the output part, all the possible products in product list and specific product features in the tree view which is also called product configuration.

There are 4 main views (figure 4.5) in the tool:

- Feature Model View
- Reference Architecture View
- Product Feature View
- Product Architecture View.

![Different views of X-MAN RAT](image)

**Figure 4.5:** Different views of X-MAN RAT
Before creating the reference architecture, it is not allowed to select the last two views. Hence, when the reference architecture is created, product list become enabled and the user can change the view to get the specific product architecture or feature model by selecting the related product from the list.

4.4 Visualizing Feature Model

When the import process is done completely, the feature model can be visualized. Feature Model View must be selected from the view list. “Feature” and “Feature Connector” buttons which are shown in figure 4.3 are pressed to visualize the feature model. When “Feature” or “Feature Connector” button is pressed, the design item creator window which is depicted in figure 4.6 is shown.

![Design Item Creator](image)

Figure 4.6: Design Item Creator

In this window, a new design item can be created for each feature by selecting a reference from the feature model and selecting a design item from the connection list. The difference between “Feature” and “Feature Connector” buttons is the feature level. That is to say that, when “Feature” button is pressed, the type of the design item will be a leaf feature. When “Feature Connector” button is pressed, the type of the design item will be a feature from the top levels in the feature tree. The only reason for that is to make the design easier for the user. Connection list in design item creator window is created according to the selected reference and feature model.
Figure 4.7: An example of a feature model in X-MAN RAT
Figure 4.7 shows a visualized feature model example. The feature model is shown in a tree view.

(a) Feature types and colours

(b) Link types and colours

Figure 4.8: Feature and link types

In addition to that there are different colours and different type of lines to express variations and commonalities. This approach will also be used when the reference architecture is created. Figure 4.8a and 4.8b show feature and link types with corresponding colours.

4.5 Creating Component-based Reference Architecture

The reference architecture is used to create products. It is instantiated and extended in compliance with the variations for the specific product in order to create the product architecture. It is an output of domain engineering process and a common asset for the application engineering process [2].

According to van der Linden, “The shared architecture in a product line is called the reference architecture. The reference architecture describes a generalised architecture that provides a solution for the range of products in the product line. It contains the variability that is instantiated in product architecting, although not all of it is visible on the architectural level [2]”.

55
Taking into consideration the definitions above, reference architecture in X-MAN RAT must:

- be compositional
- show variations
- show commonalities
- be based on a feature model
- be based on a functional model

For that reason, there must be a number of rules to create a desired architecture in X-MAN RAT. These rules are as follows:

- The elements in functional model are X-MAN components and connectors. Therefore, X-MAN RAT must use all these components and connectors to demonstrate functional and behavioural characteristics. They must be connected in the same way as in functional model.

- The features in feature model are used as a name space for the components and connectors in the reference architecture. That is to say that, when a component or connector is added to the reference architecture, a feature must be selected as a reference. Referencing means mapping as well. This mapping is not one to one. That is to say that, a feature can be used as a reference more than once. However, a component or a connector can be mapped to only one feature. The selected feature as a reference will be used in visualizing the commonalities and variations in the reference architecture and creating the product specific architecture. For example, if a feature in the feature model is mandatory, the component or connector that is referenced to that feature will be represented as mandatory in reference architecture and the specific product architecture. Similarly, if a feature in the feature model is optional, the component or connector that is referenced to that feature will be represented as optional in reference architecture. If it is selected for the specific product, it will be in the product architecture. But if it is not selected for the specific product, it will not be in the product architecture.

To create the reference architecture, components and X-MAN connectors must be added to the designer in the reference architecture view. “Component” and “X-MAN
Connector” buttons which are shown in figure 4.9 are pressed and a design item window is shown.

![Design Item Window](image)

Figure 4.9: A part of the main window when reference architecture is selected

In the design item window for the reference architecture, name is given to the components or connectors, a feature to be referenced is selected from the reference list, and the items to be connected are selected from the connection list. Figure 4.10 shows the design item window.

![Design Item Window](image)

Figure 4.10: Reference architecture design item window

Data channels are created in Data Input-Output window which is shown by selecting an item on the designer and pressing “Data View” button. As shown in figure 4.11, this form is also used to create inputs and outputs for components and connectors.
When all of the components and connectors are added according to the X-MAN RAT rules, the component-based reference architecture will be visualized. Figure 4.12 shows an example of reference architecture in X-MAN RAT.

As shown in figure 4.12, the components are represented with colours that are referenced from related features. In addition to that, the links between the components and connectors are represented with colours that are referenced from related features. The connectors are just in yellow in order to have a less complicated
In chapter 5, Home Automation System will be implemented in X-MAN RAT. Therefore, reference architecture in X-MAN RAT will be explained in more details with examples.

“Delete” button is used to delete the selected item from the architecture. “Edit” button is used to change the name of the selected item or to change the connections for the selected item.

X-MAN RAT has code functionality as well. When an item is selected and “Code View” button is pressed, compiler window which is depicted in figure 4.13 is shown.

![Compiler window for reference architecture items](image)

**Figure 4.13**: Compiler window for reference architecture items

In compiler window, the required code can be written in C# language. When “Build” button is pressed, the code is compiled. When “Run” button is pressed, the code can be tested. “Save” button helps to save the changes into the related file. If this form is shown for a selected item for the first time, there will be no source code. Otherwise, the related file will be read from the working space and be loaded to the window.
4.6 Product Features View

Product features view is used to visualize the specific product feature tree. When “Product Features View” is selected from the list, this view is activated. If a specific product is selected from product list, a feature tree is created for it. This view can be used after the visualisation of the feature model.

As shown in figure 4.14, all the features are in mandatory colour and the features which are not selected for that specific product are not included.

4.7 Product Architecture View

Product architecture view is used to create the specific product architecture. When “Product Architecture View” is selected from the list, this view is activated. If a specific product is selected from product list, product architecture is created for it. This view can be used after the creation of reference architecture.

As X-MAN RAT rules are applied, the components and connectors are referenced to the related features. If a specific feature is not selected for the product, the components and connectors referenced to that feature will not be in the product architecture.
As shown in figure 4.15, all the components are in mandatory colour and connectors are in yellow. The components and connectors which cannot exist in that specific product are not included.

Figure 4.15: An example of a product architecture in X-MAN RAT
4.8 Summary

In this chapter, X-MAN Reference Architecture Tool (X-MAN RAT) has been introduced. The main elements of the tool have been explained in details. Briefly, X-MAN RAT can import feature models in XML format. It can create a list of possible products. It can visualize feature models. It can create reference architecture based on feature model and functional model. It can visualize product specific feature trees. It can create product specific architectures. In chapter 5, X-MAN RAT will be tested with implementing a case study.
Chapter 5 – Case Study: Home Automation System

In this chapter, a case study is implemented by using X-MAN RAT to test and evaluate the tool. The selected case study is Home Automation System. This chapter starts with the description of the domain. Next, the implementation of the case study is explained step by step. Finally, the evaluation of the case study is presented.

The implementation of the case study is accomplished by following 6 steps which are as follows:

- Step 1: Create feature model in pure::variants
- Step 2: Create functional model in X-MAN component model
- Step 3: Import feature model to X-MAN RAT
- Step 4: Visualise feature model in X-MAN RAT
- Step 5: Create reference architecture in X-MAN RAT
- Step 6: Select a specific product to create product specific architecture

All these steps are explained in the next sections.

5.1 Overview of Home Automation System Domain

Home automation systems use information technology (IT) to control home appliances for many goals, most commonly convenience, energy efficiency, assisted living and security. These systems include home energy management systems (HEMS), which are networks for monitoring and controlling energy within the home. The objectives may vary, from reducing household consumption of energy and automating gadgets to potentially, in future, assisting utilities in active measures such as demand response and peak load shifting which require customer buy in. In this section, the features of the domain are explained. The feature model is inspired from the study in [52].

The features of Home Automation System are listed as follows:

- Home Automation System
  - Emergency Systems
    - Warning Systems
  - Alarm

63
Home Automation Systems mainly controls two different functionalities namely emergency systems and control systems. Emergency systems are responsible for informing related emergency services when a problem which is detected by the supervision systems occurs. Control systems are used to control lights, appliances and temperature controllers which are connected to the system. The feature model would be extended with different features. However, the main focus is on reference architecture and this feature model is adequate to create reference architecture with a considerably large number of variations.

5.2 Feature Model in pure::variants

Feature model of the domain is created in pure::variants tool. It is the part of step 1. The types of the features are listed as follows:

- Home Automation System → **Mandatory**
  - Emergency Systems → **Optional**
    - Warning Systems → **Mandatory**
      - Alarm → Or
      - Owner Call → Or
      - Emergency Call → Or
    - Supervision Systems → **Mandatory**
      - Fire → **Mandatory**
      - Intrusion → **Mandatory**
      - Flood → **Optional**
According to the preferred design, Home Automation Systems is the root feature which is mandatory. This feature has two sub-features that are Emergency Systems and Control Systems. Emergency Systems feature is optional. That is to say that, that feature may not exist in some of the products. It has two mandatory sub-features that are Warning Systems and Supervision Systems. Both of these features are mandatory because they are dependent on each other in terms of functionality. Supervision Systems use warning systems to finish their task. Warning Systems feature has three sub-features which is of or type. That is to say that, there must be at least 1 feature from Alarm, Owner Call and Emergency Call. Supervision Systems feature has two mandatory features that are Fire, Intrusion and one optional sub-feature which is Flood. Control Systems feature is mandatory. It has two mandatory sub-features that are Light Control, Appliances Control and one optional sub-feature which is Appliances Control. According to these features, the feature model can be created in pure::variants tool.

Figure 5.1: Home Automation Systems feature model in pure::variants
Figure 5.1 represents Home Automation Systems feature model in pure::variants tool. The meanings of the signs in front of the features mean as follows:

- !: Mandatory
- ?: Optional
- X: Or

Now that the feature model is designed, the XML file for the model can be created for the purpose of importing to X-MAN RAT. Figure 5.2 shows the export wizard in pure::variants tool. XML Export is chosen from the list.

![Export wizard in pure::variants](image)

Figure 5.2: Export wizard in pure::variants

When export is done, XML file of the feature model is ready. Figure 5.3 shows XML file of feature model for the case study. This file is used as input in X-MAN RAT.
Figure 5.3: Feature Model of Home Automation System in XML format

As the XML file is too big to show completely, a fragment from the XML file is shown below.

```xml
<cm:element cm:class="ps:feature" cm:default="on" cm:id="i-oa6AU5asiB1ei-2" cm:name="HomeAutomationXMANFeatures" cm:type="ps:feature">
    <cm:relations cm:class="ps:children" cm:id="iFnjlV6pcwDBBeI7">
        <cm:relation cm:id="i2pD49jJixtvNn_hS" cm:type="ps:mandatory">
            <cm:target cm:id="i9H6rTMPtwU0znnYy" />
        </cm:relation>
        <cm:relation cm:id="ip_gy9xN_f99yysTo" cm:range="[0,n]" cm:optional="true">
            <cm:target cm:id="inlxHyp84Kqpz0o6Z" />
        </cm:relation>
    </cm:relations>
    <cm:properties>
        <cm:property cm:fixed="true" cm:id="i5FOgxWpK6TyH4Zo7" cm:name="ps:Source" cm:readonly="true" cm:type="ps:string">
            <cm:constant cm:id="i3VRiZfF6FtfUjGnI" cm:type="ps:string">MERT</cm:constant>
        </cm:property>
        <cm:property cm:fixed="true" cm:id="iq_tDF5JgyXal4rE9" cm:name="ps:Created" cm:readonly="true" cm:type="ps:datetime">
            <cm:constant cm:id="iiGZDN1PuLpeGDTVA" cm:type="ps:datetime">2013-08-01T18:25:05.938Z</cm:constant>
        </cm:property>
    </cm:properties>
</cm:element>
```

As the XML file is too big to show completely, a fragment from the XML file is shown below.
5.3 Functional Model in X-MAN Component Model

Reference architecture is based on both feature model and functional model. As the feature model has been created, the functional model for the case study can be created before reference architecture design. It is the part of step 2. The functional model must represent all the behaviours for the system. By following the X-MAN component modelling rules, the components and connectors are built.

The bottom level features are the candidates for being a component for the case study. That not only makes the components easier to define but also decreases the complexity of the project as the main focus is on reference architecture. Besides, X-MAN RAT reference architecture rules must be considered when the functional model is designed.

<table>
<thead>
<tr>
<th>Component</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>InpA_T</td>
<td>-</td>
</tr>
<tr>
<td>OwnerCall</td>
<td>InpO_T</td>
<td>-</td>
</tr>
<tr>
<td>EmergencyCall</td>
<td>InpE_T</td>
<td>-</td>
</tr>
<tr>
<td>Fire</td>
<td>-</td>
<td>Out_F</td>
</tr>
<tr>
<td>Intrusion</td>
<td>-</td>
<td>Out_I</td>
</tr>
<tr>
<td>Flood</td>
<td>-</td>
<td>Out_F</td>
</tr>
<tr>
<td>LightControl</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AppliancesControl</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TemperatureControl</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RemoteControl</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: Components with inputs and outputs

Table 5.1 shows the components in the functional model with inputs and outputs. Warning systems have inputs, and supervision systems have outputs. To keep the design simple, control systems have only on and off functionalities. Therefore, control systems have no inputs or outputs. For each of the outputs of Fire, Intrusion and Flood components, there must be a data channel which is connected to all of the inputs of Alarm, OwnerCall and EmergencyCall components. RemoteControl component is used to manage LightControl, AppliancesControl and
TemperatureControl. At first glance, it seems it has no relation with any features in functional model. However, as it controls the other control systems components, it can be referenced to Control feature.

After designing of the components, the functional model can be created. Figure 5.4 shows the functional model which has been created in X-MAN GME Tool for the case study.

![Functional Model Diagram](image)

**Figure 5.4: X-MAN functional model for Home Automation System**

X-MAN functional model in figure 5.4 demonstrates the functional and behavioural characteristics of Home Automation System. Data transformations are represented as components and behaviours are represented as the control connectors. This functional model can be seen as a product which has all the functional and behavioural characteristics of the system if that product covers all the features in feature model. However, it does not show variations and commonalities.

X-MAN components and connectors shown in figure 5.4 are as follows:

- Components
o Alarm component sounds an alarm when it is triggered by fire, intrusion or flood.
o OwnerCall component calls the owner of the property when it is triggered fire, intrusion or flood.
o EmergencyCall component calls the related emergency service when it is triggered by fire, intrusion or flood.
o Fire component detects fire and the related connectors trigger Alarm, OwnerCall and EmergencyCall components.
o Intrusion component detects intrusion and the related connectors trigger Alarm, OwnerCall and EmergencyCall components.
o Flood component detects flood and the related connectors trigger Alarm, OwnerCall and EmergencyCall components.
o LightControl controls the lights with on and off functions.
o AppliancesControl controls the appliances with on and off functions.
o TemperatureControl controls the temperature with on and off functions.
o RemoteControl manages control systems. It gets the command to control LightControl, AppliancesControl and TemperatureControl components.

- Connectors
  o Sequencer_Supervision connector is a sequencer connector that connects Fire, Intrusion and Flood components. It transfers control from one component to another in a sequence.
  o Sequencer_Warning connector is a sequencer connector that connects Alarm, OwnerCall and EmergencyCall components. It transfers control from one component to another in a sequence.
  o Guard_Warning connector is a guard connector that transfer control to Sequencer_Warning connector when warning condition is true.
  o Sequencer_Emergency connector is a sequencer connector that connects Sequencer_Supervision and Guard_Warning connectors.
  o Loop_Emergency connector is a loop connector that performs the loop operation as long as warning condition is false.
  o Guard_Emergency connector is a guard connector that transfer control to Loop_Emergency connector when warning condition is true.
Selector_ControlSystems connector is a selector connector that connects LightControl, AppliancesControl and TemperatureControl components. It transfers control to the related component according to the selection condition.

Pipe_ControlSystems connector is a pipe connector that connects RemoteControl component and Selector_ControlSystems connector. It transfers control from one component to another in a sequence.

Sequencer_HomeAutomation connector is a sequencer connector that connects Guard_Emergency and Pipe_ControlSystems.

5.4 Reference Architecture in X-MAN Reference Architecture Tool

Creating reference architecture is the 5th step of the implementation. Hence, step 3 and step 4 have to be accomplished before that.

- Step 3: Import feature model to X-MAN RAT
- Step 4: Visualise feature model in X-MAN RAT

After running the X-MAN RAT, the feature model which has been exported in XML format is imported to X-MAN RAT as explained in chapter 4. When the feature model is imported, it is visualized in the feature model view. Figure 5.5 shows the feature model of the case study in X-MAN RAT.

![Figure 5.5: The feature model of Home Automation System in X-MAN RAT](image-url)
As it is expected for this case study, the number of possible products is 30. That is to say that, according to the feature types in the feature model, this software product line can produce 30 different products.

Now that the feature model and the functional model have been created, reference architecture can be designed. As explained in chapter 4, the reference architecture in X-MAN RAT must:

- be compositional
- show variations
- show commonalities
- be based on a feature model
- be based on a functional model

For that reason, a number of rules must be followed to create a desired architecture in X-MAN RAT. These rules are as follows:

- The elements in functional model are X-MAN components and connectors. Therefore, X-MAN RAT must use all these components and connectors to demonstrate functional and behavioural characteristics. They must be connected in the same way as in functional model.
- The features in feature model are used as a name space for the components and connectors in the reference architecture. That is to say that, when a component or connector is added to the reference architecture, a feature must be selected as a reference. According to this selection, the components and connectors will be in the product architecture or not.

Firstly, the table of the components, connectors and the referenced features can be constructed to demonstrate the mapping clearly. Table 5.2 shows elements from functional model and features which are referenced to them for Home Automation System.
<table>
<thead>
<tr>
<th>Component / Connector</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>Alarm</td>
</tr>
<tr>
<td>OwnerCall</td>
<td>OwnerCall</td>
</tr>
<tr>
<td>EmergencyCall</td>
<td>EmergencyCall</td>
</tr>
<tr>
<td>Fire</td>
<td>Fire</td>
</tr>
<tr>
<td>Intrusion</td>
<td>Intrusion</td>
</tr>
<tr>
<td>Flood</td>
<td>Flood</td>
</tr>
<tr>
<td>LightControl</td>
<td>LightControl</td>
</tr>
<tr>
<td>AppliancesControl</td>
<td>AppliancesControl</td>
</tr>
<tr>
<td>TemperatureControl</td>
<td>TemperatureControl</td>
</tr>
<tr>
<td>RemoteControl</td>
<td>SupervisionSystems</td>
</tr>
<tr>
<td>Sequencer_Supervision</td>
<td>SupervisionSystems</td>
</tr>
<tr>
<td>Sequencer_Warning</td>
<td>WarningSystems</td>
</tr>
<tr>
<td>Guard_Warning</td>
<td>WarningSystems</td>
</tr>
<tr>
<td>Sequencer_Emergency</td>
<td>EmergencySystems</td>
</tr>
<tr>
<td>Loop_Emergency</td>
<td>EmergencySystems</td>
</tr>
<tr>
<td>Guard_Emergency</td>
<td>EmergencySystems</td>
</tr>
<tr>
<td>Selector_ControlSystems</td>
<td>Control</td>
</tr>
<tr>
<td>Pipe_ControlSystems</td>
<td>Control</td>
</tr>
<tr>
<td>Sequencer_HomeAutomation</td>
<td>HomeAutomationXMANFeatures</td>
</tr>
<tr>
<td>Loop_HomeAutomation</td>
<td>HomeAutomationXMANFeatures</td>
</tr>
</tbody>
</table>

Table 5.2: Mapping table for components, connectors and features
The components in the X-MAN functional model can be referenced to the features that are at the bottom level. The only exception for the model is RemoteControl component. As it manages control systems and it must be in the functional model as long as these components exist in the specific product, it must be referenced to an upper level feature which is Control feature. The difficult part of referencing is in the upper levels of the feature model. The features in the upper levels are similar to the name spaces. In other words, these features can be seen as the scope of functionality. For instance, WarningSystems feature has three sub-features namely Alarm, OwnerCall and EmergencyCall. It looks like a controller for these sub-features in the feature model. In the functional model, the control connectors that are specifically responsible for the components, which are also referenced to these sub-features, are Selector_Warning and Guard_Warning. Therefore, these connectors can be referenced to WarningSystems feature as they both imply the same meaning. By taking the same approach, all referencing can be done appropriately.

Reference architecture for Home Automation System which is shown in figure 5.7 has been constructed by following the rules defined for X-MAN RAT. All of the components are X-MAN components and all of the connectors are X-MAN connectors. All of the components and connectors have been referenced to the features. All of the variations and commonalities are connected according to the feature types.

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid Black</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Dashed Black</td>
<td>Mandatory if non-mandatory parents are selected</td>
</tr>
<tr>
<td>Green</td>
<td>Optional</td>
</tr>
<tr>
<td>Red</td>
<td>Or</td>
</tr>
<tr>
<td>Yellow</td>
<td>Alternative</td>
</tr>
</tbody>
</table>

Figure 5.6: Link types and colours in X-MAN RAT Reference Architecture

The types and colours of the links are depicted in figure 5.7.
Figure 5.7: X-MAN RAT Reference Architecture for Home Automation System
The links between the design items in reference architecture shown in figure 5.7 demonstrate the relations. All of the connectors are illustrated in yellow which does not mean they are alternative in the reference architecture. If a connector has alternative type, only the link must be dashed and yellow line. However, all of the components are illustrated in their own colours which are also used in feature model view. For example, Sequencer_HomeAutomation is referenced to HomeAutomationXMANFeatures feature which is mandatory. Pipe_ControlSystem connector is referenced to Control feature which is also mandatory. For this reason, the connection between these items is a solid black line. LightControl component is referenced to LightControl feature which is mandatory. In addition to that, there is no non-mandatory parent for LightControl component. Therefore, the connection links from LightControl component to Loop_HomeAutomation connector are solid black lines. On the other hand, AppliancesControl component is referenced to AppliancesControl feature which is optional. As a result of that, the connection between AppliancesControl component and Selector_ControlSystem controller is a dashed green line as it may not exist in some of the product architectures.

When it comes to validating the five characteristics of well-structured reference architecture and rules for X-MAN RAT, it is clear that reference architecture illustrated in figure 5.6 demonstrates all these characteristics. It is compositional as it is based on X-MAN functional model. It shows variations and commonalities explicitly on components and connectors by different colours and link types. Feature model has been used for referencing the components and connectors in the architecture. Functional model has been created and all the components and connectors from the functional model have been used in the architecture. Therefore, it is based on both on feature model and functional model. Consequently, it goes without saying that the reference architecture for Home Automation System is acceptable in terms of SPLE rules.

5.5 Product Architectures

Creating specific product architecture is the last step of the implementation. X-MAN RAT creates this architecture according to the selected product from the possible products list. As there are 30 possible products for this case study and the number is too big to illustrate all the products, 3 examples will be shown in this section.
Product_1:

Figure 5.8 shows the selected features of Product_1. According to this selected features, X-MAN RAT creates the product architecture based on the reference architecture. Figure 5.9 shows the product architecture for Product_1. X-MAN RAT removes all the components and connectors from the reference architecture according to the referenced features which are not part of Product_1. The rest of the components and connectors are the mandatory parts of this product architecture.
**Product_12:**

The selected features of and the product architecture Product_12 are shown in figure 5.10 and 5.11 respectively.
Product_30:

Product_30 has all the features from the feature model. So, it has all the functionality of the system. The selected features and the product architecture of Product_30 are shown in figure 5.12 and 5.13 respectively.

Figure 5.12: Selected features of Product_30
5.6 Summary

In this chapter, the case study which is Home Automation System has been introduced and implemented in X-MAN RAT. Firstly, a feature model has been created for the domain in pure::variants tool. Next, functional model has been constructed in X-MAN GME Tool. Feature model has been imported to X-MAN RAT. Reference architecture has been created in X-MAN RAT by following the rules defined for the project. The rules were justified according to reference architecture. Lastly, product specific architectures have been created. It has been shown that creation of reference architecture is possible in X-MAN RAT.
Chapter 6 – Conclusions and Future Work

This chapter sums up the key aspects of the dissertation. First, it introduces the conclusions of the project. Lastly, it gives the future work related to the project.

6.1 Conclusions

Software Product Line Development is a development methodology that focuses on high-level reuse of large-grained software pieces. The organizations can gain many benefits by applying this approach mainly in terms of time, cost, and complexity. However, there are a number of problems that have to be resolved. There is no SPL methodology that proposes a well-defined approach to reference architecture that is one of the main problems of SPL. Component-Based Development (CBD) is another software development methodology which builds a system by bringing pre-built components together. One of the strongest aspects of CBD is reusability. CBD can be combined with software product line approach to construct reference architecture. This project brought these approaches together in order to suggest an efficient way to software product line development.

In order to get a good understanding of software product line engineering, a comprehensive background research has been carried out for the project. Chapter 2 represents that background research. It started with the general principals of SPL. Next, different aspects of Software Product Line, namely organization, requirements, feature model, functional model, reference architecture, variability management techniques, costs and benefits were explored. Finally, a number of Software Product Line projects and methodologies were investigated. From the methodologies investigated, it has been realized that it is obvious that they follow similar processes. The context analysis is the starting point in common. Domain engineering and application engineering follow the context analysis. One of the main concerns is exploiting the commonalities and variations. Nonetheless, the abstraction level is so high for all of the approaches that there is no detailed guideline to apply these approaches. These findings have been the starting point for the proposal of the new method.

Based on the literature review in chapter 2, it is obvious that there is no tool which creates reference architecture using both feature modelling and functional modelling.
FODA method is one of the well-defined approaches and base to a number of different methodologies. Hence, FODA has been selected to take part in the project. In addition to that, component-based methodologies have been investigated to combine SPL and CBSD approaches. X-MAN came forward with the advantages it provides. FODA methodology mainly focuses on the domain engineering part. However, there is no compositional or modular functional modelling approach. X-MAN component model can be used to create compositional functional model. Nevertheless, X-MAN component model cannot represent reference architecture although it can show functional model clearly. To solve these problems and create a reference architecture based on both feature models and functional models, I decided to create a new tool which is X-MAN Reference Architecture Tool (X-MAN RAT). In chapter 3, these issues have been discussed in details.

According to reference architecture definitions, reference architecture must be compositional. It must show variations and commonalities. It must be based on feature model and functional model. X-MAN RAT takes into consideration all these characteristics during reference architecture creation. In addition to that, a number of rules have been proposed to create reference architecture in X-MAN RAT. Briefly, X-MAN RAT must use all the components and connectors from functional model to demonstrate functional and behavioural characteristics and when a component or connector is added to the reference architecture, a feature must be selected as a reference. In chapter 4, X-MAN RAT has been introduced.

In order to test X-MAN RAT and prove that the suggested approach is reasonable and applicable, Home Automation System domain has been selected as the case study for the project. Chapter 5 has walked through the implementation of Home Automation System and explains how to create reference architecture and product specific architectures in X-MAN RAT based on the suggested way. A feature model has been created in pure::variants tool. The XML file of feature model has been exported from the tool. A functional model of the system has been created in X-MAN GME tool. XML file has been imported to X-MAN RAT. By following the rules defined, reference architecture for the system has been created in X-MAN RAT. Lastly, specific products have been selected from the list in X-MAN RAT and the architectures which show functionality for the specific products have been created.
Consequently, it has been shown that reference architecture in X-MAN RAT demonstrates all the characteristics of reference architecture and complies with the rules defined for the project. It is compositional as it is based on X-MAN functional model. It shows variations and commonalities explicitly on components and connectors by different colours and link types. Feature model has been used for referencing the components and connectors in the architecture. Functional model has been created and all the components and connectors from the functional model have been used in the architecture. Therefore, it is based on both on feature model and functional model. It is obvious that the reference architecture that has been constructed for Home Automation System is satisfactory in terms of SPLE rules.

6.2 Future Work

This project mainly focused on reference architecture as well as a new tool which is X-MAN RAT to create that architecture. As discussed in the previous sections, X-MAN RAT can import feature models in XML format. However, it cannot import functional models in the same way. XML files for functional models that are created in X-MAN GME Tool can be created and imported to X-MAN RAT. The integration can be established between X-MAN RAT and X-MAN GME Tool by this way. That integration can simplify the creation of reference architecture as all of the components and connectors in functional model will be created automatically. Only referencing the components and features will be carried out after importing feature model and functional model.

X-MAN RAT can work on the main types of features namely mandatory, optional, alternative and or. However, it does not support extended feature models which represents different constraints. X-MAN RAT can be extended to support these feature models as well. When these extended feature models are used, the complexity in SPL will increase dramatically. It has not been proved that X-MAN RAT can handle that complexity. I believe that it can be extended and used successfully for the extended feature models.

X-MAN RAT provides code functionality for the components and connectors in the architecture. However, the connectors in reference architecture can have generic behaviours according to their types. The user of the tool has to write the code manually for each component and connector. If X-MAN RAT is extended and
support that generic behaviour in items, it will simplify the creation of products significantly as the products will be able to be created automatically in X-MAN RAT.
Appendix

Figure A.1: Feature Model View for Home Automation System in X-MAN RAT
Figure A.2: Reference Architecture View for Home Automation System in X-MAN RAT
Figure A.3: Products Feature View for Home Automation System in X-MAN RAT
Figure A.4: Products Architecture View for Home Automation System in X-MAN RAT
References


