AN APPROACH FOR INTRODUCING A SET OF DOMAIN SPECIFIC COMPONENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İBRAHİM ONURALP YİĞİT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2015

Approval of the thesis:

AN APPROACH FOR INTRODUCING A SET OF DOMAIN SPECIFIC COMPONENTS

submitted by **İBRAHİM ONURALP YİĞİT** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Prof. Dr. Ali Hikmet Doğru
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU _____

Prof. Dr. Ali Hikmet Doğru
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Vahid Garousi
Computer Engineering Department, Hacettepe University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: İBRAHİM ONURALP YİĞİT

Signature :

ABSTRACT

AN APPROACH FOR INTRODUCING A SET OF DOMAIN SPECIFIC COMPONENTS

YİĞİT, İBRAHİM ONURALP
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ali Hikmet Doğru

September 2015, 52 pages

In this thesis, a preliminary methodology is proposed for the determination of a set of components to populate the domain model of a Software Product Line infrastructure. Software Product Line based approaches focus on the reusability of assets for a family of software products. For effective reuse, the definition of reusable assets in this thesis considers variability in a domain. The approach is based on variability specifications that is rooted in Feature Models and is reflected to a component modeling notation that addresses variability, namely VCOSEML. An initial set of components is acquired from a feature model and is modified with respect to feature constraints and design metrics corresponding to coupling, cohesion, and size oriented complexity. A component set is refined through modifications, following an iterative methodology until the developers are satisfied. The goal is to achieve a set that supports reusability – consequently to arrive at quickly converging and manageable designs through component assignments to required features. A case study is utilized in the validation of the approach.

Keywords: Software Product Line, Variability Management, Domain Specific Components, Software Reuse

ÖZ

ALANA ÖZGÜ BİLEŞEN KÜMESİNİ ORTAYA ÇIKARMAK İÇİN BİR YAKLAŞIM

YİĞİT, İBRAHİM ONURALP
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ali Hikmet Doğru

Eylül 2015 , 52 sayfa

Bu tez çalışmasında yazılım ürün hattı altyapısını oluşturacak alana özgü bileşenleri belirlemek için bir ön yaklaşım önerilmektedir. Yazılım ürün hattı, bir yazılım ürün ailesindeki varlıkların yeniden kullanımına odaklanan bir yaklaşımdır. Etkili yeniden kullanımı için yeniden kullanılabilir varlıklar alandaki değişkenlikler göz önüne alınarak belirlenmektedir. Önerilen yaklaşım, özellik modellerindeki değişkenlik tanımlamalarına ve değişkenliğin bileşen modelindeki notasyona yansıtılmasına dayanmaktadır. Ortaya çıkan ilk bileşen kümesi özellik modelinde yer alan kısıtlara ve bağımlılık, uyum, büyüklük odaklı karmaşıklık gibi tasarım ölçütlerine göre elde edilmektedir. Bileşen kümesindeki değişiklikler geliştiricilerin ihtiyaçlarını karşılayacak bir bileşen kümesi elde edinceye kadar yinelemeli bir şekilde devam etmektedir. Bu yaklaşımın amacı, yeniden kullanıma uygun, hızlıca yakınsayan ve yönetilebilir bileşen kümesine ulaşmaktır. Yaklaşımı doğrulamak için bir vaka çalışmasından yararlanılmaktadır.

Anahtar Kelimeler: Yazılım Ürün Hattı, Değişkenlik Yönetimi, Alana Özgü Bileşenler, Yeniden Kullanım

To My Family

ACKNOWLEDGMENTS

I would like thank my advisor Ali Hikmet Doğru, for his advice, motivation, patience and supervision during my research. I would offer sincere thank to my family for being supportive through all my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ACC	Average Coupling Complexity
ANMC	Average Number of Methods per Component
CASE	Computer Aided Software Engineering
CCBC	Coupling Complexity of Black Box Component
CCM	Component Complexity Metric
CCom	Cohesion Between Components
CD	Component Dependency
CFP	COSMIC Function Point
COSE	Component Oriented Software Engineering
COSECASE	COSE Computer Aided Software Engineering Tool
COSEML	COSE Modeling Language
COSMIC	Common Software Measurement International Consortium
COVAMOF	ConIPF Variability Modelling Framework
CRG	Component Relational Graph
CuCom	Coupling Between Components
FOSD	Feature Oriented Software Development
FP	Function Point
FSM	Functional Size Measurement
IC	Interface Coupling
IDE	Integrated Development Environment
IIc	Number of Incoming Interfaces
OIc	Number of Outgoing Interfaces
OVM	Orthogonal Variability Modeling
SPL	Software Product Line
SPLE	Software Product Line Engineering
TNC	Total Number of Components
TNIC	Total Number of Implemented Components
TNL	Total Number of Links

UML	Unified Modeling Language
VCOSECASE	COSECASE with Variability
VCOSEML	COSE Modeling Language with Variability

CHAPTER 1

INTRODUCTION

Nowadays, Software reuse is one of the most critical factors in improving quality and productivity [4]. From the software engineering point of view, software reuse is more efficient when systematically planned and managed for a set of applications in a specific domain. For that reason, software product family is developed for a domain, instead of creating software from scratch for each project. In this way, companies can not only quickly respond to the different requests from the customer but also put a new product on the market in a short period of time.

1.1 Overview of Software Product Lines

Companies race against each other for becoming the leader of the software market. To win the race, software developing companies must not only cope with the pace of market demands but also decrease time to market and the development costs. Thus, the researchers in the software engineering world have focused on finding new approaches in order to solve these problems. To this end, a new approach, namely Software Product Line (SPL), has been proposed inspiring of the product line concept, which is the invention of Henry Ford, in the automotive industry.

Software Product Line is a popular approach in software engineering because of development with reuse [28]. An SPL 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" [9]. The set of features specifies the particular needs of a market

segment. Software products are developed from a common set of core assets. Also, Software Product Line Engineering (SPLE) is a methodology for producing software products at lower cost, using less time and with a higher quality [37]. The crucial aim of the SPL approach is to improve reuse systematically. For successful SPLE, developing a set of core assets deals with commonality and variability among software product family [8]. Consequently, a suitable environment for reuse of core assets needs to be developed.

1.2 Variability in Software Product Lines

SPLE manages commonality (what is common) and variability (what differs between products in a family) [34]. Variability management is one of the most significant and critical issues in SPLs. There is an important difference between SPL and traditional approaches to software development. In the traditional software development, variability in time is managed in a single system [37]. On the other hand, variability in time is not enough for SPLE. Moreover, SPLE handles variability in space [37]. Both variability in time and space brings out a complicated configuration problem. Also, SPLE tries to solve the problem of configuration management.

Variability modeling defines methods that show the variability between software products in a family in order to assist engineers in work of variability management [39]. Variability in SPLs is modeled by several modeling methods such as Orthogonal Variability Modeling (OVM) [37], COVAMOF [10], and feature modeling [27]. Feature modeling is discussed in Chapter 2.

1.3 Motivation

SPL methodology empowers large scale manufacturing of a family of related software products in the software industry, and it proposes a systematic reuse strategy that covers all phases of software development. Notably, domain design phase is highlighted as a significant point to develop reusable assets. Uncontrolled variability between reusable assets in an SPL finishes off the potential benefits of domain commonal-

ity and decreases the effectiveness of domain variability handling [28]. Therefore, variability in a domain must be taken into account while defining reusable assets. Otherwise, it is clear that unproductive systematic reuse damages the advantages of SPL approach and threatens the overall quality of an SPL.

Definition of domain specific components is a critical aspect of SPLE. However, the existing methodologies do not present a systematic way of how to define the set of domain specific components. In this thesis, we propose a primary methodology for the determination of a set of components to populate the domain model of an SPL infrastructure. This research presents an approach based on a well-defined set of principles, guidelines, and metrics. Moreover, the proposed approach makes it possible to exploit the commonality and manage variability in both problem and solution space. A case study is utilized in the validation phase of the approach, as well.

The motivation of this thesis is to construct a procedure to define domain specific components for SPLs. The main contribution of the study is to propose a preliminary approach to determine a set of components to populate the domain model of an SPL infrastructure. The objectives of this thesis study can be summarized as follows:

1. Explore how to manage variability in the existing SPL methodologies.
2. Create a new tool to model a set of components with representing variability.
3. Propose a new methodology to define a set of components for a specific domain.
4. Evaluate a determined set of components in terms of reusability.

1.4 Related Works

There are some similarities between this study and the work of Ileri et. al. [25] in which they introduce a component-based variability modeling approach to manage variability in a component model with Component Relational Graphs (CRG). In addition, their approach handles commonality and variability among components within a given domain, in terms of both problem and solution spaces. On the other hand, the study comes up with a solution for only the component variability. The proposed ap-

proach does not provide a guideline for the determination of components in a product line. Also, tool support for variability representation is left as a future work.

In the study of Asikainen et. al. [1], a similar approach for modeling software product families is presented. The approach introduced by Koalish, which is a modeling language that extends Koala [44], is a component model and architecture. Koalish provides a method to deal with modeling and configuring components and interfaces, by taking the variability concept into account. Also, this study provides a variability resolution technique at compile time. Nevertheless, the approach does not have a useful mechanism for modeling composition variability.

In the work of Haber et. al. [23], a hierarchical approach is introduced to manage variability by extending MontiArc architectural description language. This study defines a variability modeling approach that supports the component-based systems. MontiArc represents the component variability with a hierarchical structure. Moreover, this study provides a tool support for hierarchical modeling. Although the tool has the capability to be modeled in solution space, this study is planned to support the modeling problem with space variability as a future work. Also, this approach lacks in handling variability in components and composition among components.

In another study conducted by Bayraktar, a method for representing variability in components is proposed by relating them with the OVM, which provides a separate model for managing variability [5]. The method introduces an approach to manage variability information for different configuration of products and trace variability between versions. Moreover, the tool supports OVM in the scope of this study. Therefore, this method is appropriate for component-based software systems to create software product family models. Nevertheless, the method does not address a solution for defining an appropriate set of domain specific components.

In a recent study by Balci, a new method is proposed to create a reference architecture for component-based SPLE [2]. The study focused not only on creating component-based reference architecture but also managing variability among components in the infrastructure of SPL. Also, a new tool which is called X-MAN Reference Architecture Tool is developed for constructing a reference architecture. Although this tool works well for handling the main types of features, it does not support variability

constraints among features.

1.5 Thesis Organization

In Chapter 2, some background information is given to provide a better understanding of the main topic of this thesis study. The proposed solution for the definition of domain specific components is explained in Chapter 3. In Chapter 4, a case study conducted in cloud management domain will be explained. The experimental study to evaluate the approach is discussed in Chapter 5. Finally, conclusion and future work is expressed in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, the fundamental concepts which provide a basis for this thesis are studied in detail. First of all, Software Reuse is introduced. Secondly, Software Product Line Engineering and its general principles are examined. After that, "Variability Management" and "Variability Modeling" terms in Software Product Line Engineering are given details. Finally, software metrics at design level is briefly explained.

2.1 Software Reuse

Software reuse is the concept of developing software using existing software pieces instead of building from scratch [32]. Software reuse is not limited to using only the source code of the previously developed software while building a different software. All assets that arise during the software development process can be converted into reusable assets [38]. Software architecture, requirements, design, test descriptions, and documents are examples of reusable assets in software projects [3, 6].

In recent years, scope and complexity of the software-intensive systems have grown dramatically. Software companies have been in constant struggle to find better ways to decrease time to market, cost and effort of the software development. Software reuse is the most reasonable solution to this problem. Software companies utilize software reusability approaches, for the simple reason that software reuse provides many advantages for developing large scale software systems. As stated in [36], these advantages can be summarized as follows:

- Reduce effort, time and cost of development,
- Improve productivity and quality of products,
- Increase reliability of products,
- Decrease maintenance cost.

Consequently, software reuse is an attractive approach for many organizations.

Experience indicates that there are five phases of software reuse for an organization [21] (See Figure 2.1). Each phase increases the reusability with respect to its predecessor stage. As a result, upper level run phase of software reuse corresponds to increased benefits in terms of time to market, cost, and quality because of the improved usage rate of reusable assets. Increased level of software reuse not only accelerates the product development stage, but also reduces development costs and increases end product quality [20].

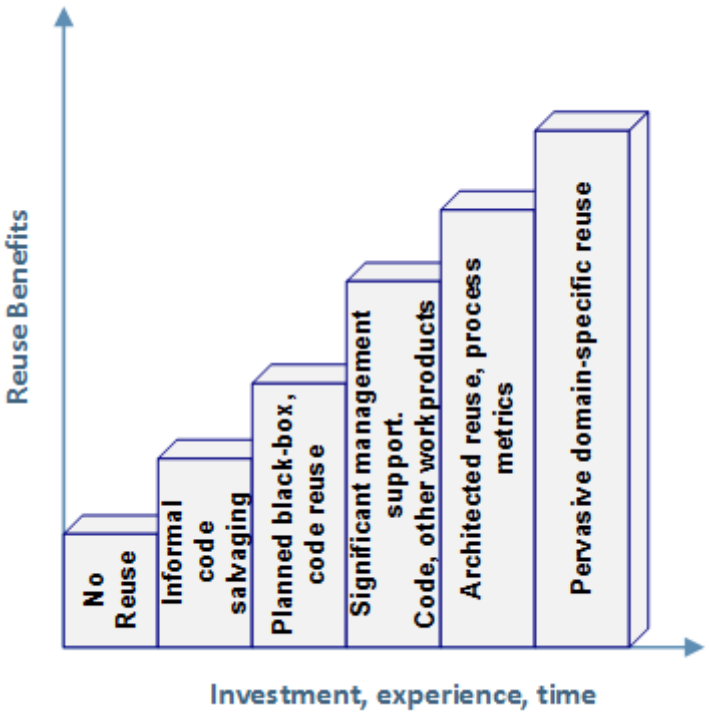

Figure 2.1: The incremental adoption of software reuse: modified from [21]

"Pervasive domain-specific reuse" is seen to be the most effective software reuse level, when the adoption levels of software reuse are examined [14]. The scope of the domain should be set out clearly in order to reach the highest level software reuse.

Hence, the domain is analyzed for discovering the commonality and variability in the domain. After the domain analysis, domain specific architecture and components are defined by considering the commonality and variability in the domain. Hereupon software products are being developed in accordance with the domain specific architecture by using domain specific components.

There are two types of software reuse; opportunistic reuse and systematic reuse. Opportunistic reuse defines that new software is developed from existing software. However, existing software is modified to satisfy variable customer needs of the new software [31]. Systematic reuse provides the guarantee of software reuse through descriptions, responsibilities and tasks for a particular organization [19]. In systematic reuse, a new software is developed to utilize reusable assets such as architecture, and components. Moreover, reusable assets can enhance the profit of software reuse. Today, opportunistic reuse is replaced with systematic reuse. Organizations prefer systematic reuse to opportunistic reuse because systematic reuse brings more benefits than opportunistic reuse.

2.2 Software Product Line Engineering

Nowadays, large-scale software intensive systems are being developed as a product family instead of a single product. The product family brings flexibility to meet various customer requirements. Also, it reduces development time for producing new products. Software Product Line is a popular approach regarding product family in software engineering due to the advantages that it offers to software development time, cost and effort [22]. The main purpose of SPLs is to meet quickly variable customer/market requirements. Moreover, SPLs intend to develop particular products at lower cost and with less effort.

Software Product Line Engineering (SPLE) is a methodology that leads to high-level reuse with a systematical way. SPLE handles commonality and variability among software products for the product family [43]. SPLE makes use of domain knowledge to exploit commonality and variability between the products for creating reusable elements. Reusable elements in an SPL are named as “core assets” that are particular to

a domain. Core assets that generate the skeleton of an SPL are created by considering commonality and variability in the domain. Besides, SPLE emphasizes the fact that new product must be developed by using the core assets.

SPLE is composed of two phases: Domain Engineering and Application Engineering for reuse and development with reuse [42]. As shown in Figure 2.2, Domain Engineering is a phase that is composed of domain analysis, domain design, domain realization, and domain testing processes in order to develop core assets. All of the core assets constitute the infrastructure of the product line. On the other hand, Application Engineering briefly defines a development process where software products are produced from reusable core assets. Application Engineering consists of four main processes, namely; application requirement engineering, application design, application realization, and application testing. As a summary, Application Engineering phase yields the end products that are built on the core assets which are created in Domain Engineering phase. SPLE does not consist of only two primary phases. SPLE

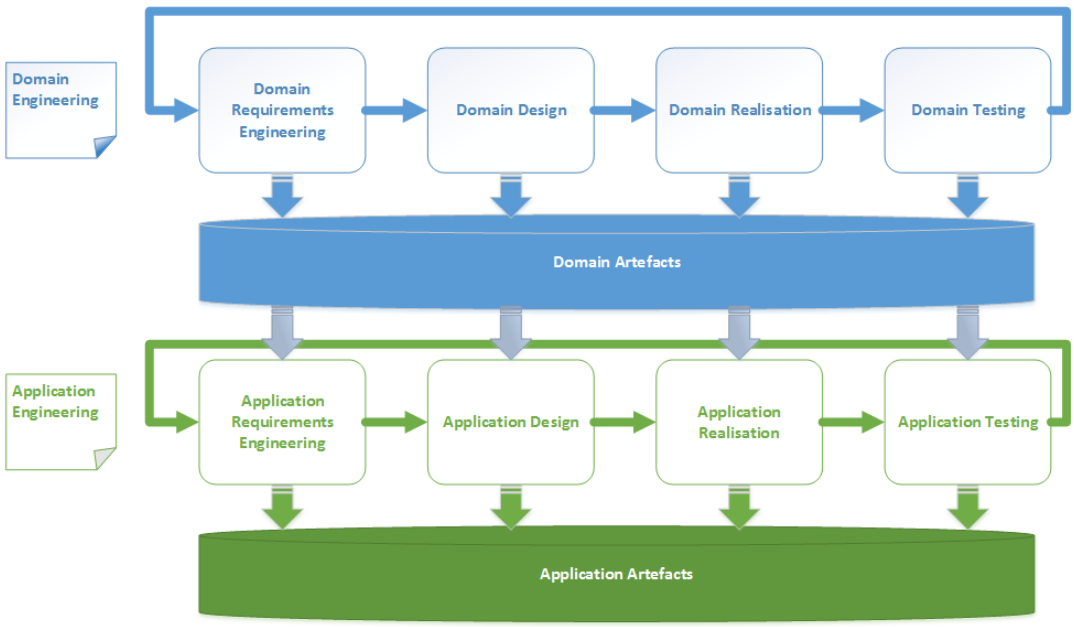

Figure 2.2: SPLE phases (adapted from [37])

describes three essential activities: Core Asset Development, Product Development, and Management. Figure 2.3 represents these three essential activities of SPLE. Core Asset Development is corresponds to Domain Engineering Phase, whereas Product Development is equivalent to Application Engineering phase. Management is also an

essential activity to manage the development activities in both technical and organizational point of view. If the roles and responsibilities are appropriate to administer the organization aspect of SPLE, the organization must be successful in the application of SPL [42]. Furthermore, the activities affect each other, and they arise from interaction [35].

Figure 2.3: Essential activities of SPLE according to [35]

2.3 Variability Management in Software Product Line Engineering

Variability defines how different products can be changed or customized in SPLs and is the center element of SPLE [43]. Variation points and variants recognize the variability of SPL [2]. There are two types of variations, which are variation in time and space [24]. Variation in time is associated with the specific products that have different versions. On the other hand, variability in space is identified by two particular products in a family with variants which is present in various releases.

Variability management is about handling the introduction, use, and evolution of variability [39]. Managing variability among software products of a family is one of the most significant activity in SPLE. Variability between particular products of a family must be administered in a systematical way. Moreover, variability management

covers all development processes from requirement engineering to testing. However, increasing the number of variations among the products is in parallel with the enlarging size and complexity of the products. Therefore, variability management turns into a complicated task in order to deal with commonality and variability in a product line.

Variability management consists of the set of activities that are used to determine and organize commonality and variability in product families. SPLE deals with emerging domain specific products, taking commonality and variability into consideration [8]. For developing software products by using core assets, domain engineers must detect and manage variability between software products.

2.4 Variability Modeling

Variability modeling is a method to represent variability among software products of a domain. The concept not only provides an understanding of variability but also assists domain engineers in the task of variability management [39]. Variability modeling expresses common and variable parts in SPLs. Moreover, it defines constraints between variable parts explicitly. Variability management plays a significant role in both phases of SPLE, domain engineering and application engineering. Consequently, variability modeling should be employed for both phases. There are numerous approaches for variability modeling such as feature modeling, COVAMOF [10], and OVM [37]. Feature modeling is used in the scope of this thesis. For this reason, feature modeling is discussed in the following paragraphs.

A feature describes a software property and represents functionality in a domain [26]. A feature model as represented in a hierarchical tree structure is a composition of mandatory, optional, and alternative features [26]. Also, feature model is a powerful notion of handling the complexity in SPLs. From the end user view, the model provides a presentation of the characteristics in a domain. Essentially, it consists of the common features and variations of software products in a domain. Thus, variability modeling with feature modeling is one of the key strategies for indicating the problem space of SPLs.

There are four groups of feature types that categorize the association between a parent

feature and a child feature. These feature types are explained as follows:

- *Mandatory* represents the features that must be in every possible configuration of particular products.
- *Optional* demonstrates that the features can either be or not be a part of the specific product.
- *Or* signifies that at least one of the child features must be selected for the particular products.
- *Alternative* displays that only one of the child feature has to be selected for the product derivation.

Variation points and variants express feature variability. A variation point defines what can vary in a feature [37]. A variant specifies how a feature can vary [37]. Every variation point has a set of related variants that determine how the variation point can be resolved [23].

Feature variability also defines two kinds of constraints between features: requires and excludes [18].

- **Requires constraint:** If a feature A is selected to be a part of a configuration, then also feature B has to be selected.
- **Excludes constraint:** If a feature A is selected to be a part of a configuration, then feature B cannot be selected.

Figure 2.4 represents a case of feature model with the description and the relationship between features. Commonalities are displayed using "mandatory" and "or" features in a domain while variability is represented via not only "optional" but also "alternative" features.

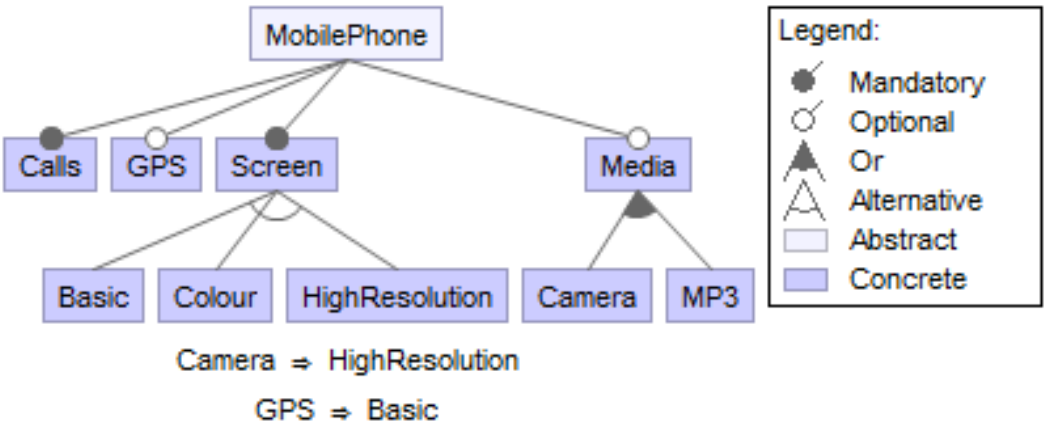

Figure 2.4: An example feature model (from [7])

2.5 Software Metrics at Design Level

Software measurement is one of the crucial points in Software Engineering in order to provide clear information about the achievement of a determined goal. Software estimation effort is a series of actions to forecast the size of a software product in order to achieve the desired result [15]. Also, it is utilized as a part of the estimation that people need to devote some time on for software projects to be successful. Decision makers must be acquainted with the exact size of a software product to control and organize the software development life cycle.

Software metrics play a significant role in detecting errors at the design level of software development life cycle. The detected errors can be prevented at an early stage of software development process thanks to software measurements at the design level. In this thesis, we suggest that the determined set of domain specific components should be evaluated in terms of complexity size, coupling and cohesion metrics. Therefore, software metrics at design level will be shortly introduced.

2.5.1 COSMIC Functional Size Measurement Method

COSMIC is a method, which was founded by The Common Software Measurement International Consortium (COSMIC) group in 1998, to measure the functional size of a component [13]. The COSMIC group proposed a new method to measure the

functional size of a software based on functional user requirements [17]. The new method can be utilized in not only embedded but also enterprise software products [16]. The studies indicate the suitability of COSMIC method for measuring functional component size at the design level [15].

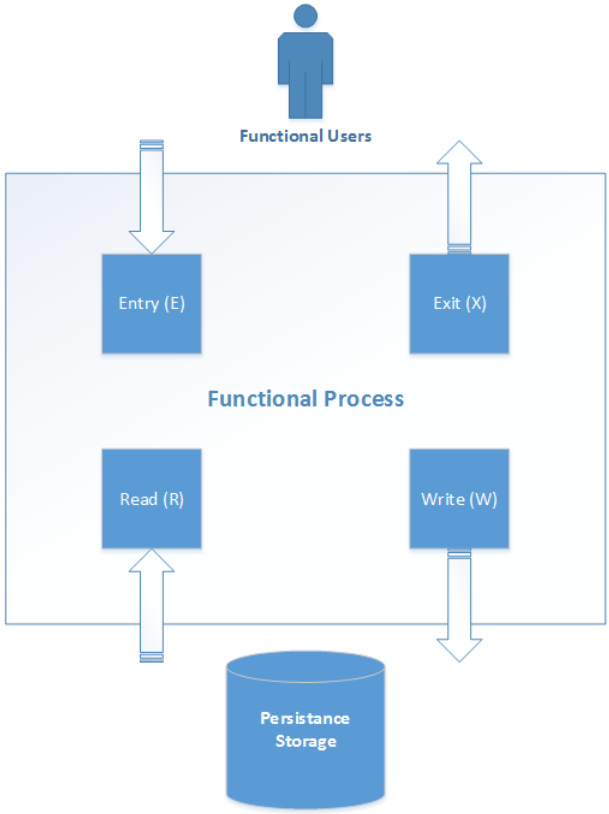

Figure 2.5: Data movements of COSMIC method (from [15])

Data movements of COSMIC method are represented in Figure 2.5. Each data movement is equal to 1 COSMIC Function Point (CFP). Data movement types are explained as follows:

- **Entry (E):** The data group flows from the functional users beyond the boundary into the functional process layer.
- **Exit (X):** The data group flows from the functional process beyond the boundary to the functional users.
- **Read (R):** The data group flows from the persistence storage to the functional process.
- **Write (W):** The data group flows from the functional process into the persistence storage.

The data movements are counted for each functional process in order to calculate CFP. The CFP of each data movement types is summed up to get the size of the functional process. As the result of this calculation, the total function size is computed for a software product.

$$FunctionalSize(process)i = \sum Ei + \sum Xi + \sum Ri \sum Wi \tag{2.1}$$

2.5.2 Coupling and Cohesion Metrics

It is preferred to have components which are less dependent on each other as much as possible. In order to reach high quality and ease of the maintenance software products, software engineers should devise a component set according to two perspectives that are coupling and cohesion. Therefore, the determined set of components should be evaluated considering the low coupling and high cohesion principles.

Coupling between components (CuCom) is a design metric to calculate coupling for each component [45]. CuCom indicates the dependency of a component to other components.

$$CuCom = CD + IC \tag{2.2}$$

CuCom: Coupling between components

CD: Component dependency

IC: Interface coupling, which is the number of inflows of a component

Coupling complexity of a black box component and average coupling complexity are other design metrics to measure coupling for a component based system [33]. These design metrics are given as follows:

$$CCBC = IIc + OIc \tag{2.3}$$

$$ACC = \sum_{i=1}^{n} \frac{CCBCi}{n} \tag{2.4}$$

CCBC: Coupling complexity of a black box component

IIc: Number of incoming or required interfaces

OIc: Number of outgoing or required interfaces

ACC: Average coupling complexity

n: Number of components

In the study of Yadav et. al. [45], Cohesion between components (CCom) is a design metric that can be utilized to compute cohesion for each component in the set. CCom metric points out the relevancy between the components. We are following the cohesion concept as introduced in this study.

$$CCom = \begin{cases} CCM = 0 & \text{if } CC = TIC \\ CCM = 1 & \text{if } CC = 0 \\ \sum_{i=1}^{n} \frac{CCi}{TIC} & \text{Otherwise} \end{cases} \tag{2.5}$$

TIC: Total number of interfaces between other components

CC: Number of caller components

CCM: Component complexity metric

n: Number of components

2.5.3 Complexity Metrics

Complexity metrics can be used to measure and evaluate the design model composed of domain specific components. The metrics provide a way to predict both maintainability and integrability of software product line infrastructure at the design level. The structural complexity of each component depends not only on the functional size but also on the attributes such as the set of components, connectors between components, interfaces of each component, and composition tree [30].

There are two complexity metrics for predicting maintenance and integration efforts at design level. One of them is Correction Effort per Component in order to measure component maintainability [40]. The following regression model can be used to

calculate maintenance effort for a set of components.

$$\text{Correction Effort} = \exp(\text{-0.02*TNC + 0.07*TNIC + 0.02*ANMC - 2.37}) \quad (2.6)$$

TNC: Total number of components

TNIC: Total number if implemented components

ANMC: Average number of methods per component

Another complexity metric is Integration Effort to compute the total effort spent on integration among components while building the end software product [40]. The following regression model can be utilized to measure integration effort for a component-based product line using only the number of links between components.

$$\text{Integration Effort} = 0.1 * \text{TNL} + 2.6 \quad (2.7)$$

TNL: Total number of links

CHAPTER 3

PROPOSED SOLUTION

In this chapter, the proposed component modeling language for variability management, basic information about the tool selected for feature modeling and the new tool created for a component model are explained. Finally, the proposed solution is discussed in detail.

3.1 VCOSEML

Component is an element of software, as a building block that contains complex functionalities. Component-based software systems are developed by selecting appropriate components and assembling those together [11]. Component Oriented Software Engineering Modeling Language (COSEML) is a graphical and proper modeling language for visualizing a hierarchical decomposition [12]. Different from the "Component Based" approaches, components are the fundamental notions starting with the abstractions in requirements and continuing through the executable code. COSEML is supported with a graphical tool, modeling a system as components and their connections. COSEML begins with the abstraction of system parts to introduce the building blocks of a system. After that physical components need to be corresponded to encapsulate the detailed functionalities in the defined abstract modules [25]. Links connect abstractions to physical components. Table 3.1 explains further detail about the graphical primitives of COSEML.

Table 3.1: COSEML symbols and their meanings (from [11])

Symbol	Explanation
	**Package:** Package is for organizing the part-whole relations. A container that wraps system-level entities and functions etc. at a decomposition node. Can contain further Package, Data, Function, and Control elements. Also can own one port of one or more connectors. Can be represented by a Component.
	**Function:** Function represents a system-level function. Can contain further Function, Data, and Package elements. Can own connector ports. Can be represented by a Component.
	**Data:** Data represents a system-level entity. Can contain further Data, Function, and Package elements. Can own connector ports. Has its internal operations. Can be represented by a Component.
	**Control:** Control corresponds to a state machine within a Package. Meant for managing the event traffic at the Package boundary, to affect the state transitions.
	**Connector:** Connector represents data and control flows across the system modules. Cannot be contained in one module because two ports will be used by different modules. Ports correspond to interfaces at components level.
	**Component:** A Component corresponds to the existing implemented component codes. Contains one or more interfaces. Can contain components. Can represent abstraction.
	**Interface:** An Interface is the connection point of a Component. Services requested from a component have to be invoked through this interface.
	**Represents:** A Represents relation indicates that an abstraction will be implemented by a Component.

Abstractions in COSEML are Package, Data, Control and Function at the logical level. Packages not only refer to the abstract components but also encapsulate abstraction groups associated elements. Also, the Package are detailed through further Package, Data, Control, and Function abstractions. Besides, components and their interfaces are main elements at the physical level. Connectors represent communications among components, as well as between abstractions. Both logical and physical level elements in COSEML provide an opportunity to represent the solution in order to find out the subproblem of the entire system. Therefore, COSEML is capable of representing not only the solution space but also the problem space for design-

ing component-oriented software systems. Consequently, system requirements are transformed into a set of components and connectors between components.

Unfortunately, COSEML does not support variability management. Moreover, a model of COSEML does not point out which elements are variable and how the variability is realized. Therefore, it becomes a requirement to find new approaches or improve COSEML to manage variability at the design level. Variability modeling has been introduced to component-oriented development by importing from OVM and Feature Model. Hence, COSEML with variability (VCOSEML) is introduced. VCOSEML is a representation technique specified for variability modeling [46]. Figure 3.1 displays specific variability symbols of VCOSEML.

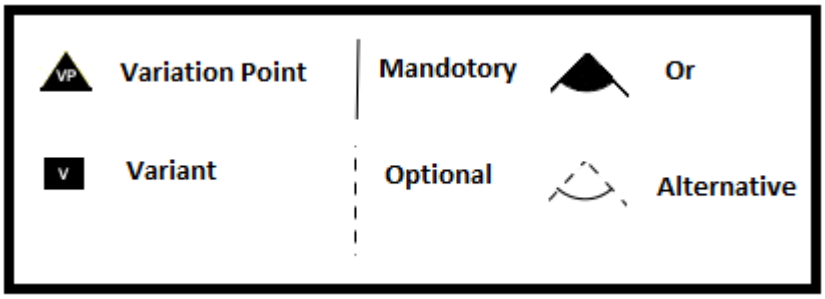

Figure 3.1: VCOSEML variability symbols

Abstract primitives in VCOSEML are used for problem space modeling, where as components and connectors are used for solution space modeling. Thus, variability can be tracked not only in the problem space but also in the solution space through VCOSEML. Furthermore, VCOSEML essentially focuses on the hierarchical decomposition of a domain as well as variability management.

Figure 3.2 shows an example component model of VCOSEML. In this example, the problem space of the domain is modeled through abstraction elements. Also, the solution space is designed by using components and connectors that represent communications among components. This modeling view provides the general aspect of the set of components for the purpose of developing software products with domain specific components in an SPL. Moreover, it decreases complexity in managing the variability in an SPL through assisting the domain engineers in their analysis for changing effects for component selection.

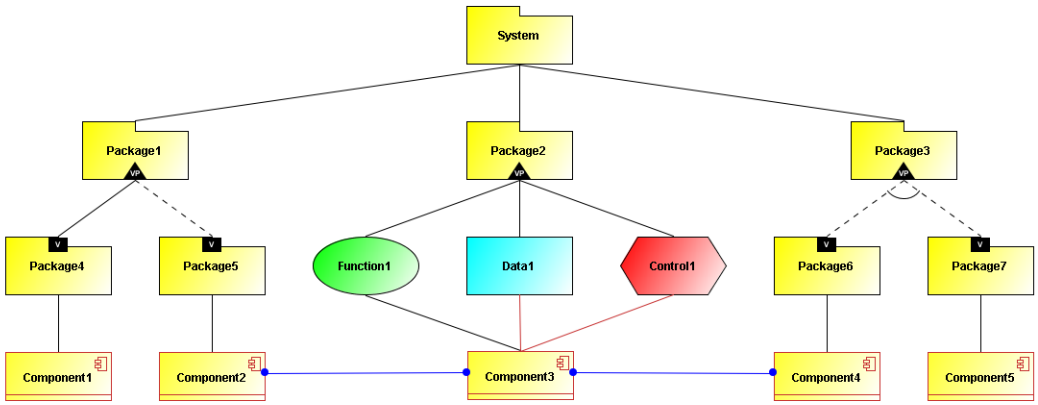

Figure 3.2: Example component model of VCOSEML

3.2 Tool Support

Before going into details about the proposed approach, it is helpful to provide some preliminary information about the tool support in this study. Feature IDE was used for feature modeling and VCOSECASE was implemented to manage variability when defining domain specific components in an SPL infrastructure. The rest of this section contains an introductory overview about these tools to accomplish the goals of this study.

3.2.1 FeatureIDE

FeatureIDE is an open source tool for Feature Oriented Software Development [29]. In this study, a feature modeling tool is a necessity in order to analyze the domain. FeatureIDE is selected because of its simple and friendly user interfaces for modeling a feature diagram. Also, the tool provides textual modeling besides graphical user interfaces. Furthermore, both Domain and Application Engineering phases of SPLE are powered by FeatureIDE. Therefore, it is used to create and manage the feature model in the scope of the study. A screenshot is given from FeatureIDE in Figure 3.3.

FeatureIDE is able to visualize feature models in a hierarchical tree format. Features in the model can be managed using the graphical user interfaces of the tool. The feature model created by using FeatureIDE is contained in all associations and constraints between features. For this reason, the tool can be utilized to handle variability

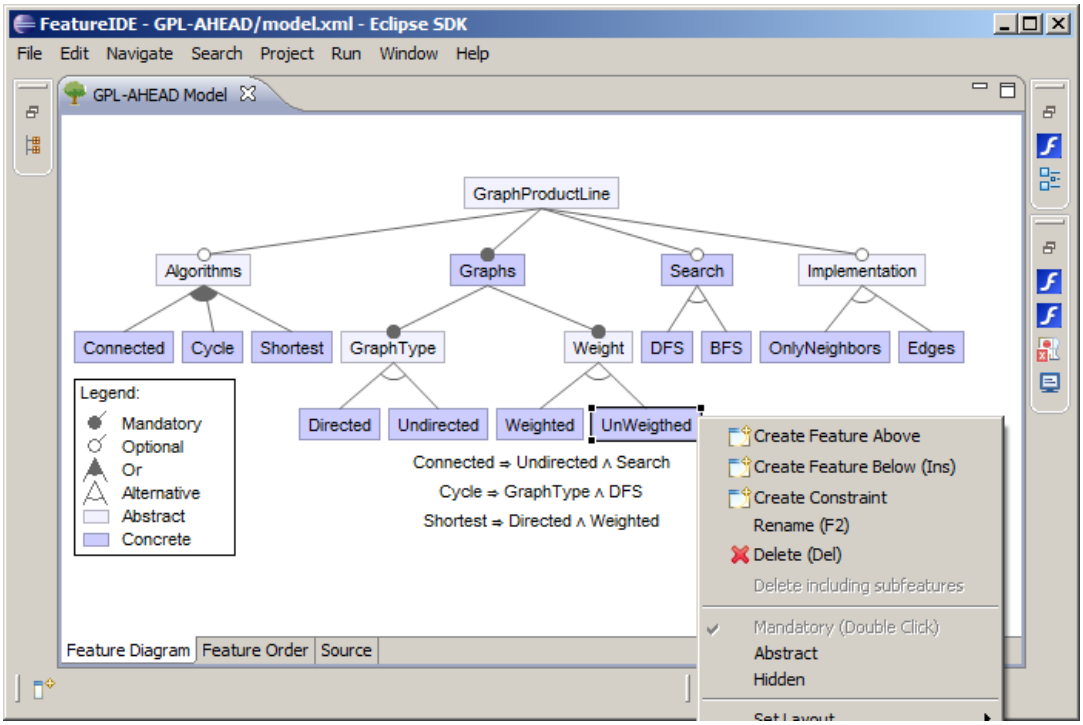

Figure 3.3: FeatureIDE screenshot that displays a feature model (from [41])

and constraints on the feature model.

3.2.2 VCOSECASE

This study also covers the improvements to COSECASE, which is the graphical tool for supporting COSEML. COSECASE does not support graphical representation of variability management in either logical or physical level. It brings the need of a tool support to create a component model with variability in a domain. Thus, the tool was enhanced to support the functionalities about variability management at both logical and physical level. The new version of COSECASE is called VCOSECASE, as it is basically COSECASE with variability. In this section, newly added functionalities for managing variability at the design level are explained.

VCOSECASE provides domain engineers to use not only all COSEML elements but also VCOSEML items in both logical and physical levels. It is a CASE tool to model easily the structural views of the domain. Domain engineers are able to decompose hierarchically a domain in accordance with domain requirements and graphically rep-

resent the structural views of the domain architecture composed of domain specific components. Interaction between domain specific components is also modeled with the tool using the hierarchical diagram.

VCOSECASE has features for managing variability in a domain. The functionalities about variability management in an SPL is brought with the tool. VCOSECASE is defined after adding the new features. Some of the new features that support variability management are:

- Creating a new VCOSEML model for designing domain reference architecture,
- Integrating variation points and variants in a domain design model,
- Viewing variability dependencies between VCOSEML elements,
- Adding variability constraints for each VCOSEML element,
- Making modifications in VCOSEML elements.

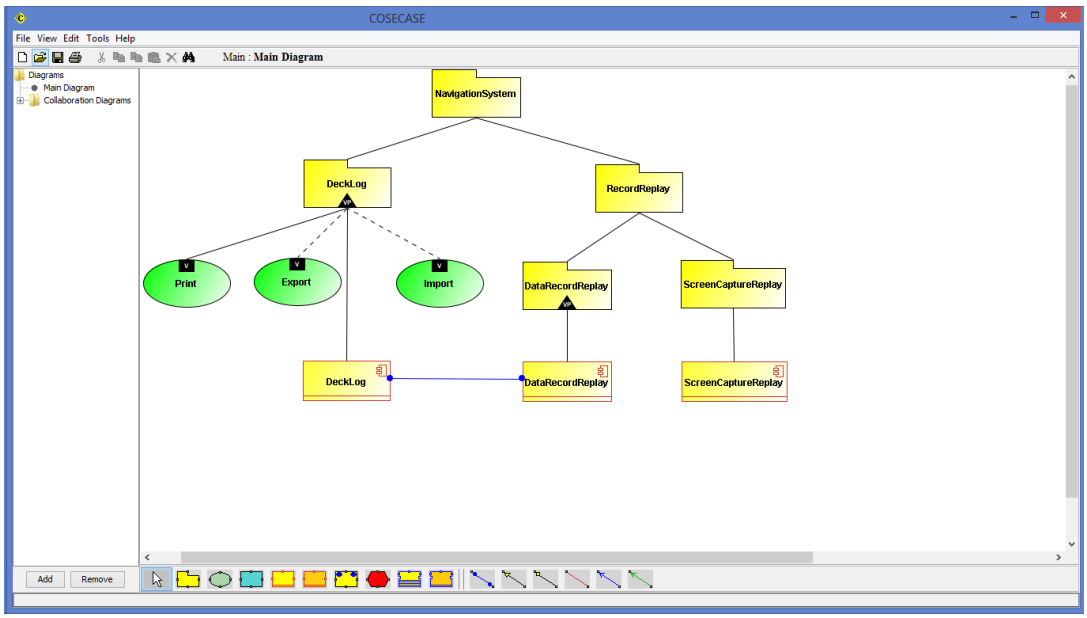

Figure 3.4: An example screen of VCOSECASE

VCOSECASE is capable of managing variability in both problem and solution spaces. The user manual for using variability features in VCOSECASE is introduced in Appendix A.

3.3 Definition of Domain Specific Components Approach

Domain design is highlighted as a crucial concept to develop reusable and flexible domain specific components. The main purpose of domain design is to identify domain specific components and interaction between them. The set of domain specific components must be reusable and flexible because all software products of the family are developed using these components.

The proposed solution is an introductory approach to build and evolve a reusable infrastructure, which covers all variability requirements contained in an SPL. The approach is based on the structural decomposition of a domain and variability specifications that is rooted in feature models and reflected a variable component model represented with VCOSEML. An initial set of proposed components is modified with respect to the variability constraints and design metrics that regard coupling, cohesion, and size oriented complexity. The modification of the component set has been continued iteratively until an optimal set of components is achieved. Iterations are excepted to improve the set of components, to reflect at the design metrics. However, when to stop the iterations is a decision to be made by the domain designer.

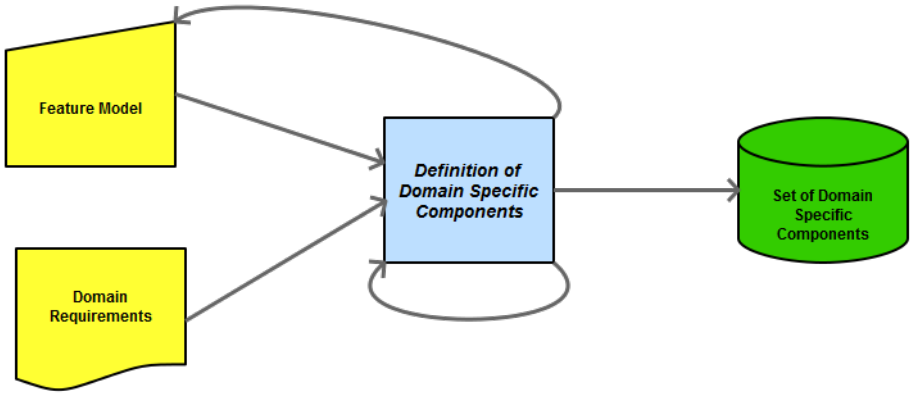

Figure 3.5: Overview of the proposed approach

The following steps are suggested in defining domain specific components:

1. Map feature model to abstractions,
2. Embed feature variability in abstractions,
3. Define domain specific components,
4. Evaluate and revise the component set.

An overview of the proposed approach can be observed in Figure 3.5. The proposed approach steps are explained in the rest of this section.

3.3.1 Map Feature Model to Abstractions

The transition process from the feature model in the domain analysis to the definition of domain specific components begins with the creation of a model of VCOSEML. The model is created to map the feature model to the abstractions of VCOSEML. The mapping process from features to abstractions is straightforward and is a one-to-one relationship between features and abstractions. Domain engineers manually assign features to abstraction elements of VCOSEML. The mapping process is carried out with the tool that can help domain engineers to understand the complex design of SPLs.

Domain engineers use abstractions of VCOSEML such as Package, Function and Data for modeling the system graphically. Package abstraction illustrates subparts of a system at the logical level. Function and Data abstractions represent the details for each subpart of a system.

3.3.2 Embed Feature Variability in Abstractions

We consider that feature variability should be traceable through the abstractions of VCOSEML model in the design of components. Determined feature variability during domain analysis are embedded in the abstractions through the variability elements of VCOSEML. Variation points, variants, and variability constraints in a feature model can be reflected to a VCOSEML model. Thanks to the visualization abilities of VCOSEML, feature variability can be traced at the design level.

3.3.3 Define Domain Specific Components

The relationship between abstractions and components are established at this step. While abstractions reside in the problem space, realizations of the abstractions are part of the solution space. The abstractions are implemented by components at the

physical level. A top-down approach is followed for defining components in the domain. The components that correspond to the abstractions are determined by applying the following design rules:

1. *Each variation point in the abstraction corresponds to at least one component.*

   The system should be designed such that at least one component corresponds to a single variation point. Figure 3.6 illustrates an example of this case. Packages that contain variability points are implemented as separate components.

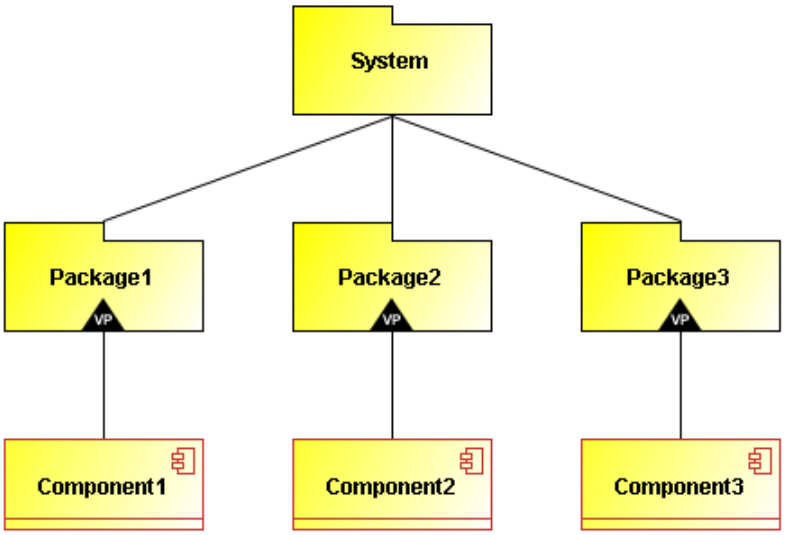

Figure 3.6: An example case for design rule 1

2. *If a variant excludes another variant at the same variation point, variants will be mapped to separate components.*

   Managing mutually exclusive variables is one of the common problems in SPLE. There should be no 'excludes' variability constraints between the variability elements in the scope of the corresponding component.The activation of a group of features at the same time may cause the system to not work properly. Therefore, it is recommended to implement the components separately whenever a variant excludes another variant at the same variation point.

   Figure 3.7 shows an example of expressing this case. Although Package4 and Package5 are at the same variation point, they correspond to two separate components because the variants of the packages are mutually exclusive. In case

that the variant of Package6 and the variant of Package7 are not mutually exclusive, single component covers the variants.

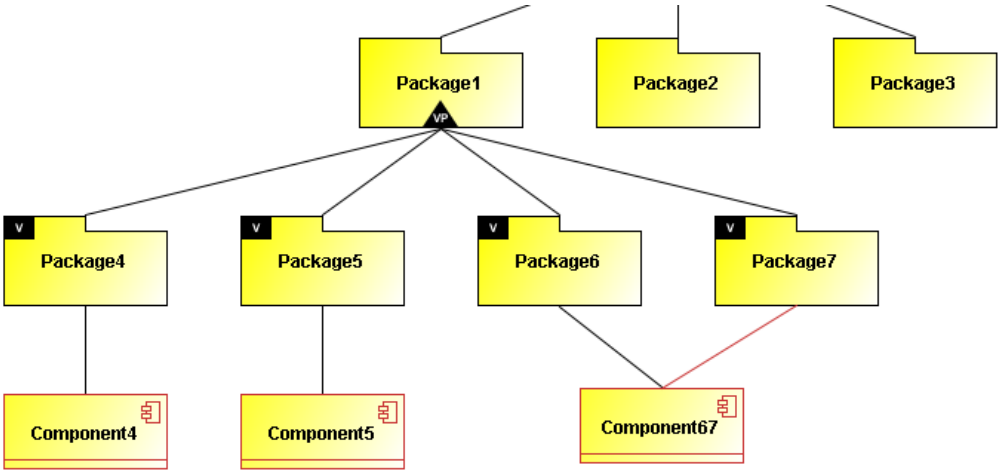

Figure 3.7: An example case for design rule 2

3. *If a variant is an alternative to another variant at the same variation point, variants will be mapped to separate components.*

   Variants which are alternative to each other are mutually exclusive as if design rule 2. Thus, it is suggested that the variants are implemented as individual components. This case is depicted in Figure 3.8. Package4 and Package5 correspond to individual components. Because of this, this package contains variants that are alternative to each other.

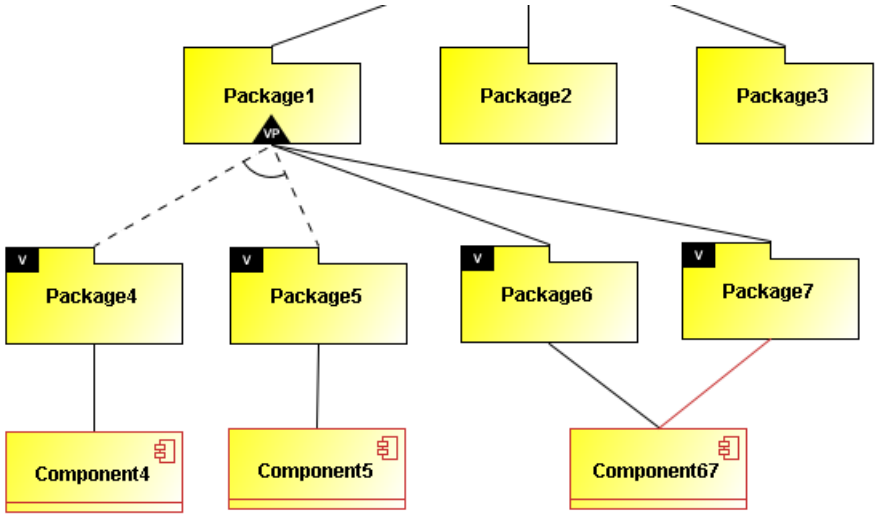

Figure 3.8: An example case for design rule 3

4. *If a variant requires another variant at the same variation point, variants will be mapped to the same component.*

   The variants that require each other at the same variation points are implemented by the same component in order to adapt low coupling and high cohesion design principles. The example given in Figure 3.9 illustrates that Package6 and Package7 are mapped to the same component for the reason that the variants of these packages require each other.

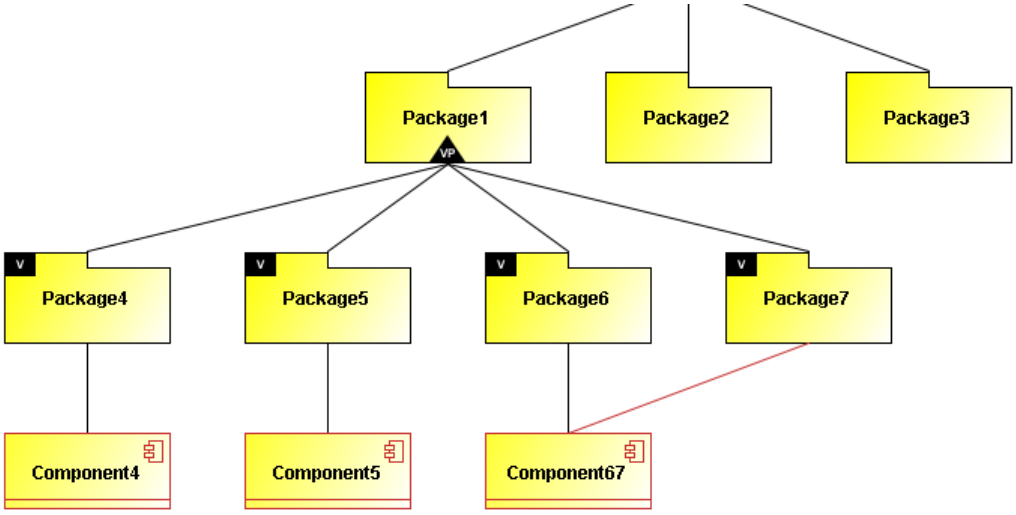

Figure 3.9: An example case for design rule 4

5. *If a variant requires another variant at a different variation point, variants will be mapped to different components, which are linked with connectors.*

   In the design rule 1, it is recommended that each variation point correspond to at least one component. If the variants between the variation points require each other, the components that implement the variants are linked with connectors in order to communicate with them. An instance of this case is given in Figure 3.10. Component1 and Component2 are connected with a connector whereas Function1 and Data1 include in the variants that require each other at different variation points.

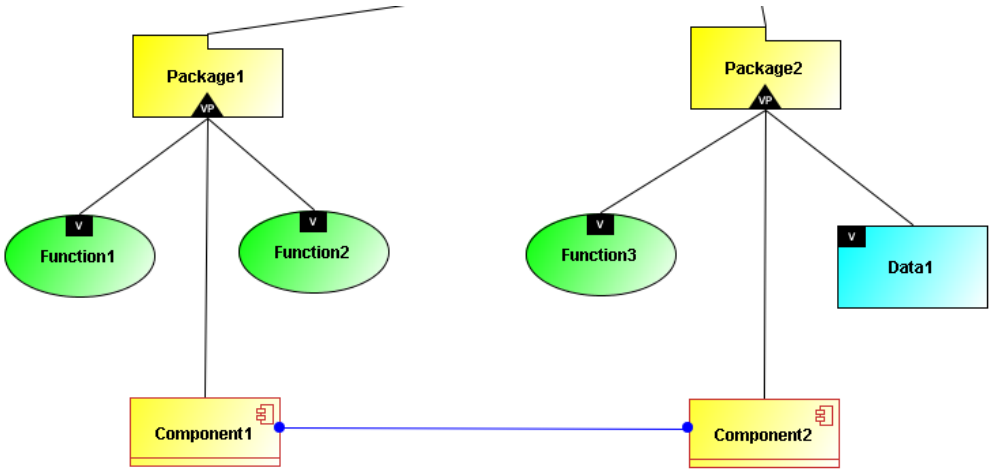

Figure 3.10: An example case for design rule 5

6. *If variants excluding each other require another variant at the same variation point, the required variant will be mapped to the individual component that links with other components by using connectors.*

   In the design rule 2, it is suggested that the variants correspond to individual components whenever a variant excludes another variant at the same variation point. The design rule 6 is proposed such that the variant which is required by variants excluding each other at the same variation point is linked with other components by using connectors. Also, the component must be linked with other components by using connectors. Thus, the parts that are dependent on other components excluding each other are minimized. Moreover, by shrinking the dependent parts is intended to increase the number of common components which can get along well with other components.

   An example case is given in Figure 3.11. Although both the variants of Package4 and Package5 are mutually exclusive, they require the variant of Package6. For that reason, the packages are implemented by separate components. Finally, Component6 is linked with Component4 and Component5 because of the required variant that Component6 contains.

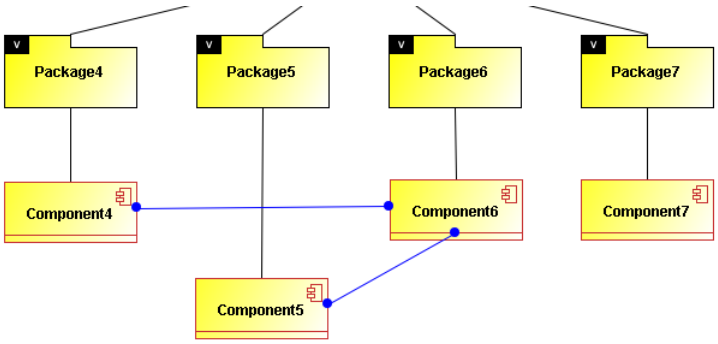

Figure 3.11: An example case for design rule 6

7. *If a variant is optional, it will be mapped to the component that links with other components, which contain mandatory variants at the same variation point, by using connectors.*

   The design rule 7 is proposed such that the abstractions that are defined as optional are implemented separately from the mandatory component. In case that optional variables are selected, the optional components are also selected. Optional and mandatory components should be connected via connectors. The example of this case in Figure 3.12 illustrates how the components are connected using connectors.

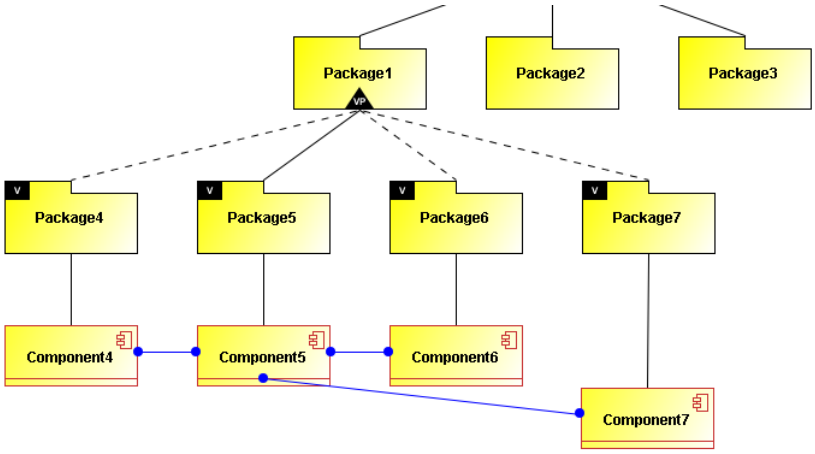

Figure 3.12: An example case for design rule 7

8. *If a variant at different variation point is required by at least two variants at different variation points, the variant will be mapped to the individual component that links with other components.* Different variants under different variation points are implemented by different components. If the variants require each

other, the connectors provide the interaction between the components. According to the design rule 8, a variant at different variation point corresponds to the individual component. The component is linked with separate components in the case that 'requires' constraints are available from at least two variants at different variation points.

Figure 3.13 displays an example case for expressing the design rule 8. Package3 and Package6 are implemented by individual components because of that the variants are at different variation points. On the other hand, Function1 is implemented by Component1 and connected the other component through the connectors for the reason that the variants require the variant in Function1.

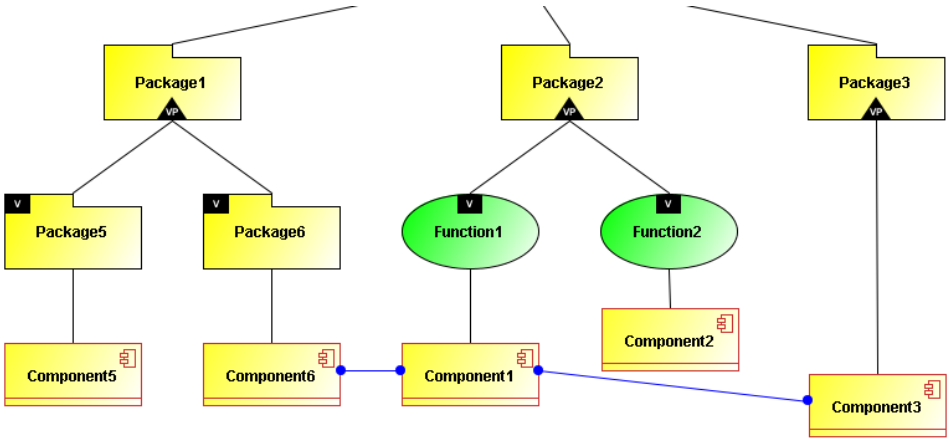

Figure 3.13: An example case for design rule 8

3.3.4 Evaluate the Set of Domain Specific Components

The approach suggests that the determined set of domain specific components should be measured in accordance with software metrics at the design level. Hence, the result of measurements is evaluated in terms of component size, coupling, cohesion and structural complexity. Also, the set of components is appraised using complexity metrics in order to predict maintenance and integration efforts. First of all, the set of components is reviewed in the direction of the results of measurements. Components having the above average complexity and dependence on other components are detected. After considering the outputs about the evaluations, the component set can be modified by reapplying the recommended rules of the approach. This process might be performed in numerous iterations until an optimal set of components is reached.

CHAPTER 4

CASE STUDY

In this chapter, a case study on cloud management domain is performed, to test and evaluate the proposed solution presented in this thesis. This chapter begins with the overview of cloud management domain. After that, the proposed approach is explained step by step for the case study under consideration. Finally, the results of this case study are given.

4.1 Overview of Cloud Management Domain

In this section, we conducted a case study to investigate the validation of the approach.

Figure 4.1: Overview of cloud management domain

The case study involves a product family of the cloud management domain. The software products in the domain provide not only cloud monitoring but also resource planning for managing cloud resources. Also, it is planned to optimize the utilization of cloud resources with respect to their performance, cost and energy consumption.

Domain analysis was conducted before applying the approach. Domain requirements and feature model were prepared during the domain analysis process. The feature model of the cloud management domain is created in FeatureIDE tool. The feature model is shown in Figure 4.2.

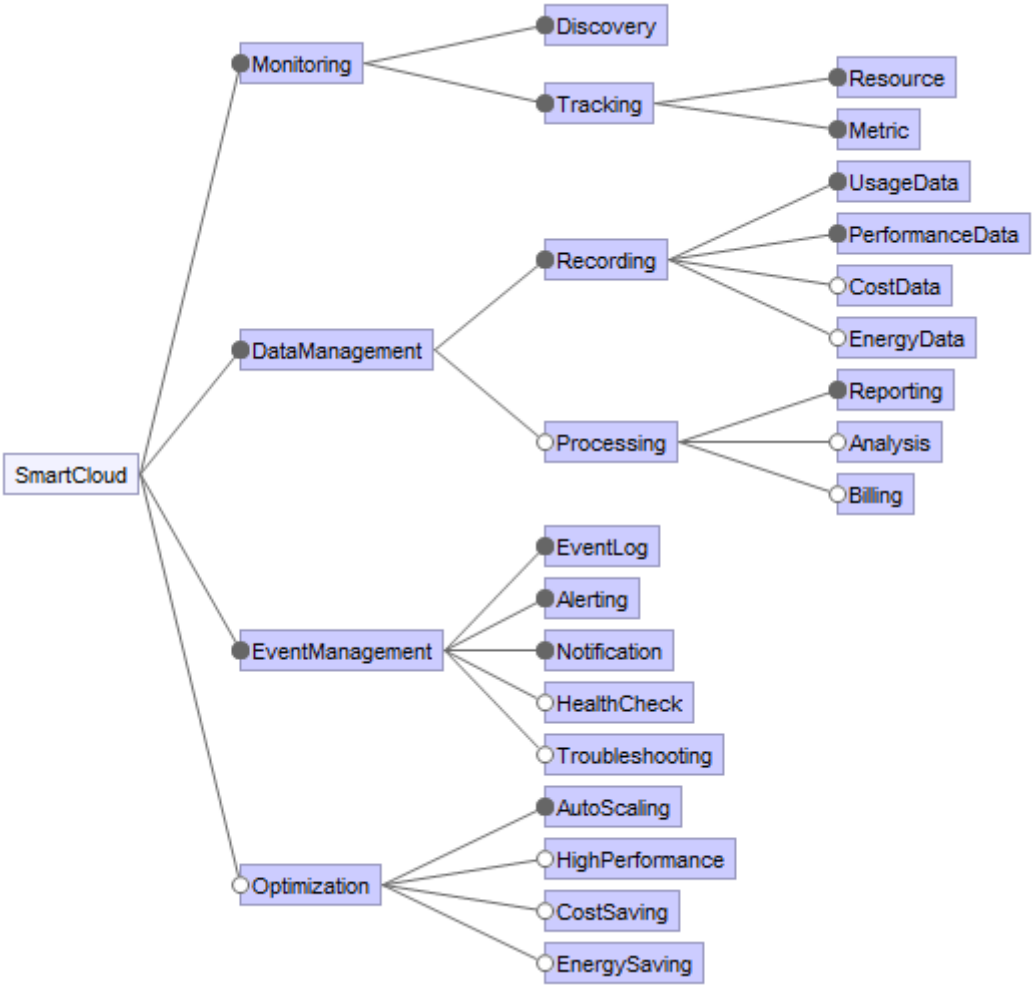

Figure 4.2: Feature model of cloud management system

4.2 Definition of Component Set in Cloud Management Domain

In the first stage of the approach, the feature model was mapped directly to a model of VCOSEML through the abstractions. After mapping the features to the abstractions, feature variability (variation points, variants and variability constraints) was embedded in the VCOSEML model.

Figure 4.3 displays the first level abstractions of the cloud management system using VCOSEML.

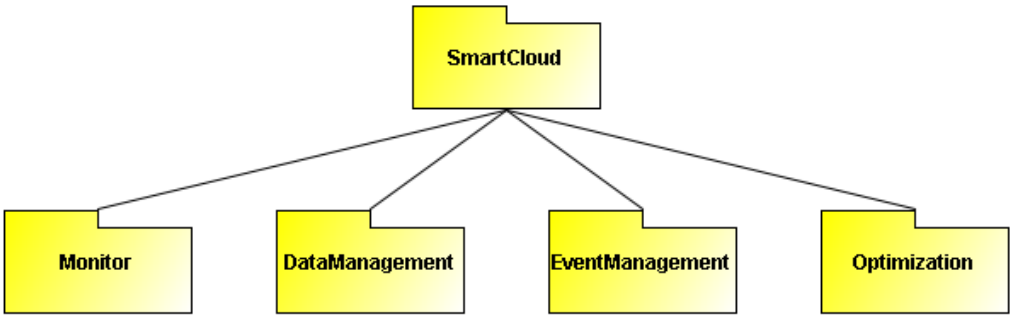

Figure 4.3: First level abstraction of the cloud management system

Afterward we continued the next step of the approach that is the definition of domain specific components. By this way, the set of components for cloud management domain was determined by following the recommended rules. Furthermore, the structural view of cloud management system was expressed through the VCOSEML model that consists of abstractions and their matching components. The following figures represent the big picture emerged as the result of the first iteration.

Figure 4.4 displays the decomposition of the *Monitor* package, which is in charge of discovering and tracking cloud resources.

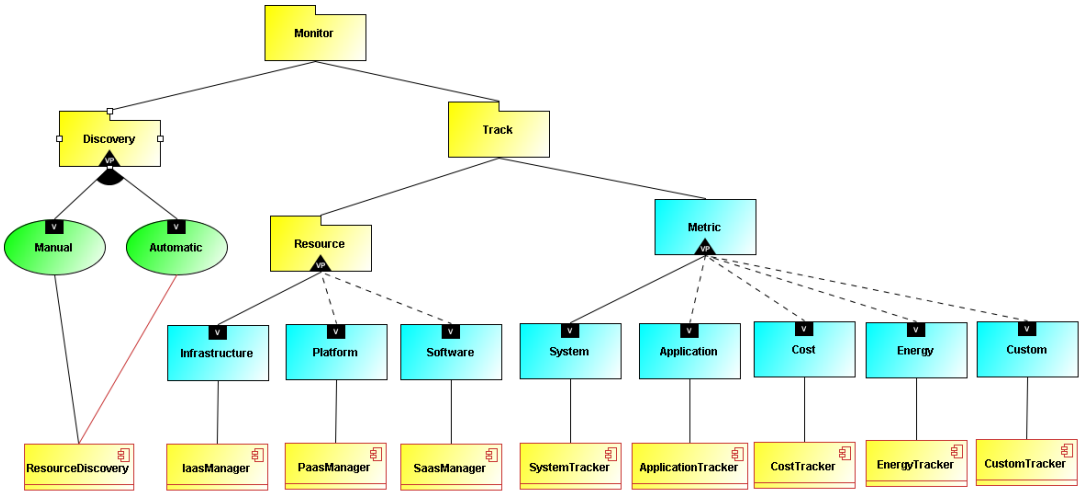

Figure 4.4: Decomposition of Monitor package

Figure 4.5 represent the decomposition of the *DataManagement* package, which accounts for recording and processing data from cloud resources due to report, analyze, and bill.

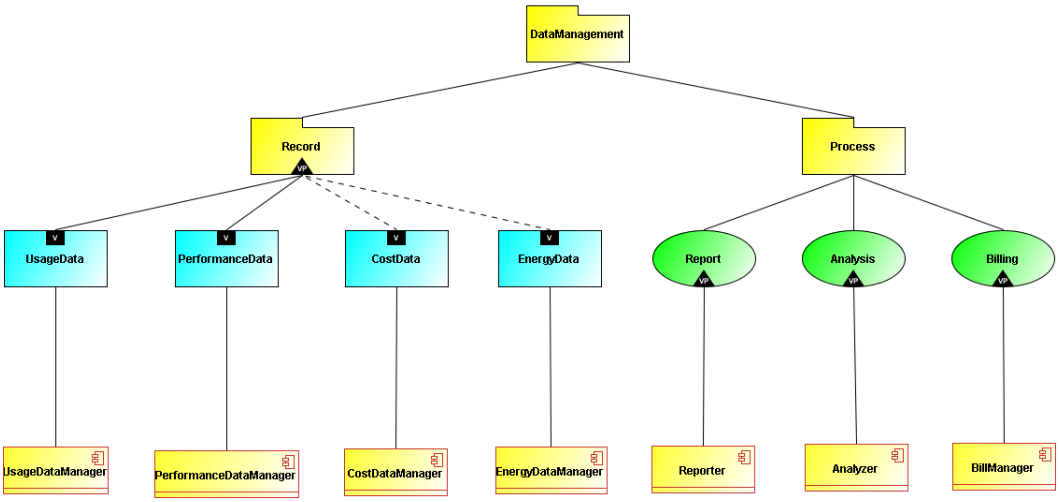

Figure 4.5: Decomposition of DataManagement package

The decomposition of the *EventManagement* package is shown in Figure 4.6. The package handles logging events, notifying problems and recommending solutions to problems.

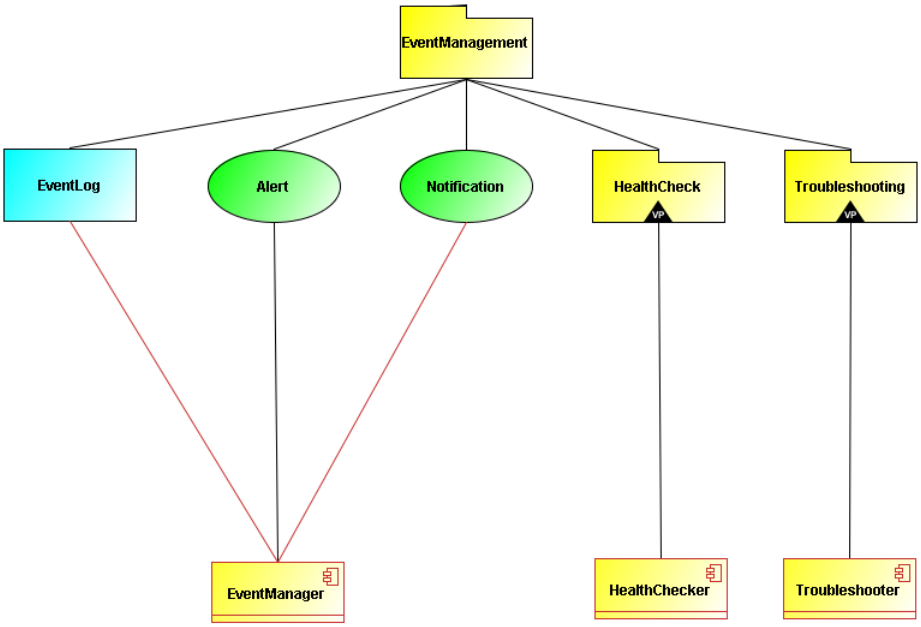

Figure 4.6: Decomposition of EventManagement package

In Figure 4.7, It can be seen the decomposition of the *Optimization* package. The package optimizes cloud resources in terms of usage, performance, cost, and energy.

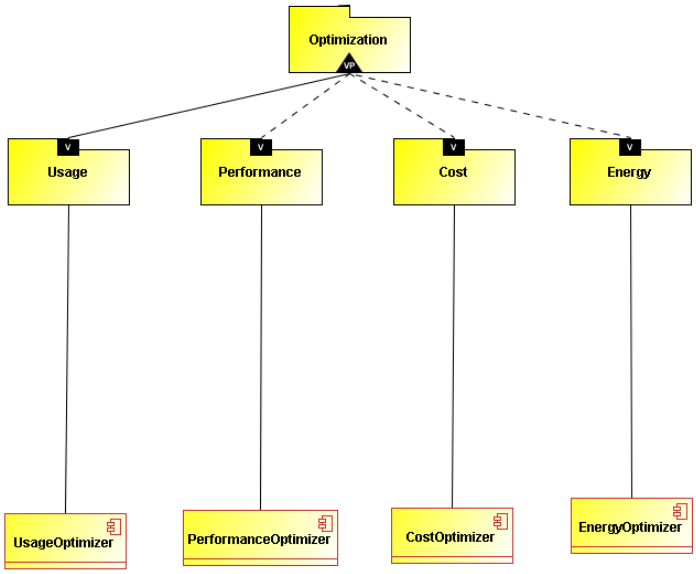

Figure 4.7: Decomposition of Optimization package

4.3 Evaluation of Component Set

The defined domain specific components were measured in accordance with software metrics at the design level. The measurement results were evaluated in terms of complexity size, coupling, and cohesion. Moreover, the set of components was appraised according to complexity metrics in order to predict the efforts spent on both maintenance and integration. After the evaluation of measurement results, the set of domain specific components was modified with respect to the design metric until the optimal set of components were acquired.

When the first iteration was completed, 23 components were determined for the cloud management domain. After the components were determined, the functional size of each component is manually computed using COSMIC FSM method. The calculated function points of the components obtained from this case study are tabulated in Table 4.1.

Furthermore, the dependency of other components are calculated for each component in terms of coupling and cohesion. The component relationship diagram can be referenced to calculate coupling and cohesion between the components at the design level. The arrows among the components in the diagram presented in Figure 4.8 illustrate the relations. All arrows demonstrate the dependency among the set of components.

Table 4.1: Results of COSMIC FSM method

Component	**Entry (E)**	**Exit (X)**	**Read (R)**	**Write (W)**	**FP**
ResourceDiscovery	9	5	-	12	26
IaasManager	12	8	8	12	40
PaasManager	8	8	8	5	29
SaasManager	8	8	6	3	25
SystemTracker	18	-	-	17	35
ApplicationTracker	20	-	-	18	38
CostTracker	16	-	-	16	32
EnergyTracker	16	-	-	16	32
CustomTracker	21	-	-	20	41
UsageDataManager	8	16	18	-	42
PerformanceDataManager	9	18	18	-	47
CostDataManager	8	15	15	-	38
EnergyDataManager	8	15	15	-	38
Reporter	16	8	20	6	50
Analyzer	18	30	32	7	87
BillManager	10	6	14	6	36
EventManager	18	28	18	24	88
HealthChecker	8	16	8	8	40
TroubleShooter	14	14	14	-	42
UsageOptimizer	24	14	-	12	50
PerformanceOptimizer	20	8	-	8	36
CostOptimizer	12	4	-	4	20
EnergyOptimizer	12	4	-	4	20

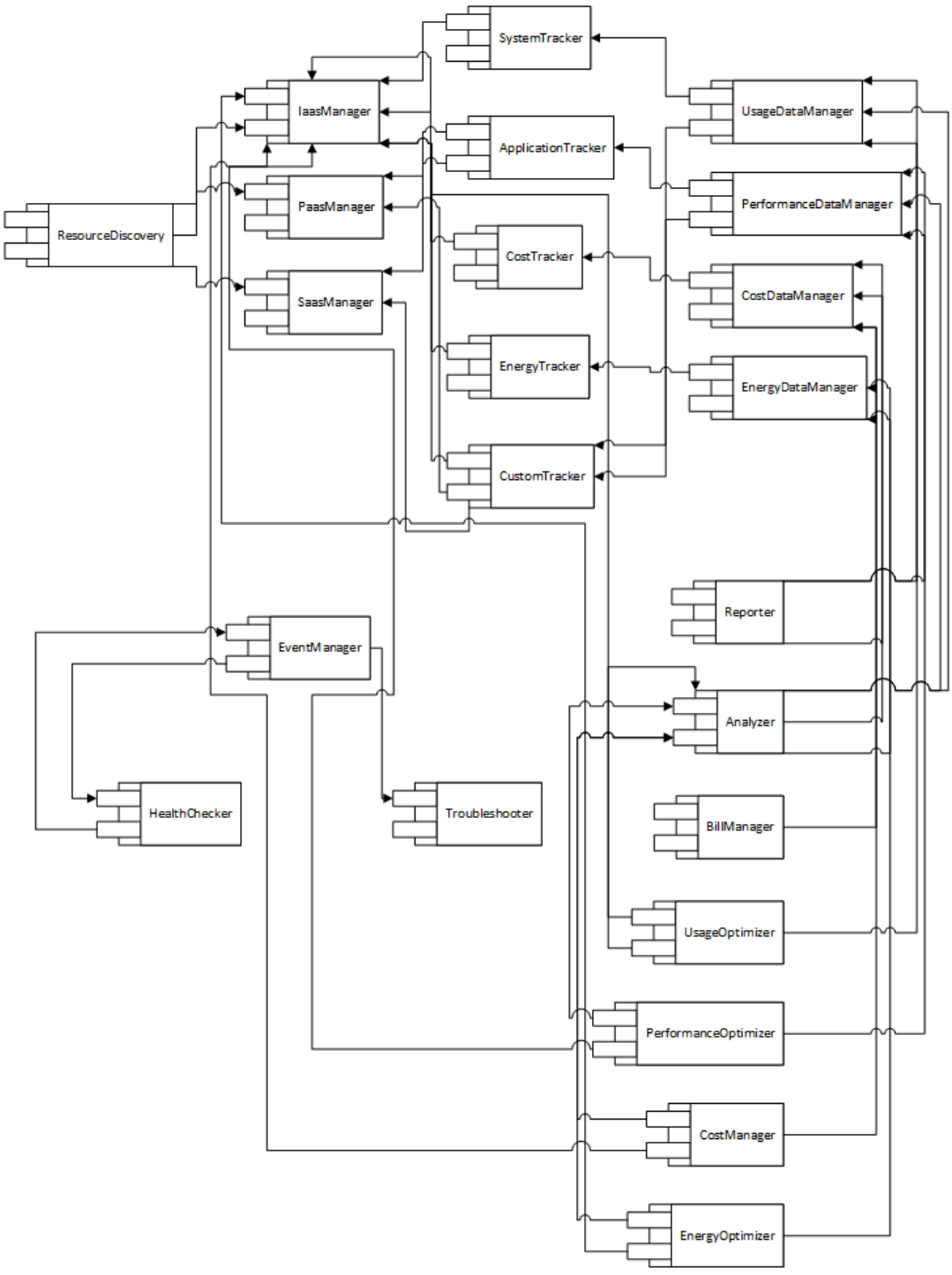

Figure 4.8: Component relationship diagram of cloud management system

The results of the first iteration in accordance with coupling and cohesion design metrics are given in Table 4.2.

Table 4.2: Dependency measurement results

Component	Coupling	Cohesion
ResourceDiscovery	3	1
IaasManager	5	0.86
PaasManager	3	0.75
SaasManager	3	0.75
SystemTracker	2	0.66
ApplicationTracker	3	0.5
CostTracker	2	0.66
EnergyTracker	2	0.66
CustomTracker	5	0.57
UsageDataManager	5	0.80
PerformanceDataManager	4	0.75
CostDataManager	5	0.80
EnergyDataManager	3	0.75
Reporter	2	1
Analyzer	8	0.44
BillManager	1	1
EventManager	7	0.43
HealthChecker	2	0.66
Troubleshooter	1	1
UsageOptimizer	3	1
PerformanceOptimizer	3	1
CostOptimizer	3	1
EnergyOptimizer	3	1

As can be observed in Table 4.1 and Table 4.2, components called Analyzer and EventManager are not behaving like other components. When we analyzed the outputs of the first iteration, we detected these anomalous components in terms of not only complexity size, but also dependency on other components. Therefore, the feature model was reviewed for expanding the features realized by the determined components. Moreover, the feature variabilities are updated according to the emerging features.

After updating the feature model and feature variabilities, we continued with the second iteration. In the second iteration, the Analyzer was noticed to require to be de-

composed into four subcomponents (PerformanceAnalyzer, CostAnalyzer, TrendAnalyzer, WhatIfAnalyzer). Also, EventManager was divided into three subcomponents (EventLogger, AlertManager, NotificationManager). At the end of the second iteration, we reached a more appropriate set of components in terms of reusability.

On the other hand, the results of two iterations were compared with regards to maintainability and integrability. Correction Effort per component was equal to 195.195 for the set of components in the first iteration. Otherwise, the result for the same metric was 41.347 in the second iteration. Also, Integration Effort in the second iteration was equal to 6.6 while the result was 6.3 in the first iteration. Although the results of iterations were almost equal according to Integration Effort metric, the result obtained in the second iteration was much better that the result of the first iteration in terms of maintainability. Therefore, we also achieved a more maintainable set of components at the end of the second iteration.

CHAPTER 5

DISCUSSION

There are some problems to be considered in this study, which might be improved in this approach. Firstly, the proposed approach supports only feature variability in the feature model. Nevertheless, variability management should expand through all the stages. Hence, the approach should be enhanced to support the different life cycle stages of SPLE in the future. Secondly, variability constraints between the items of VCOSEML are not shown graphically in the model. Even though constraints are not represented in the model, the tool provides a way to keep the textual information about variability constraints between elements. Finally, it was not possible to make a comparison of our results with the results of other approaches because there are no existing approaches in the literature. Updated versions of the approach might be compared to former versions. Thus, improvements of the approach can be evaluated, and these evaluations will provide significant information for further research.

The direction of the study was to propose a preliminary methodology in order to define a set of components to populate the domain model of a Software Product Line infrastructure. In the light of this motivation, the following goals are set in the scope of this thesis.

- Offering a systematic way to define the set of domain specific components
- Introducing a well-defined set of principles, guidelines, and metrics
- Exploiting the commonality and manage variability in both problem and solution space

Despite the existence of these issues, the case study illustrates to accomplish the goals of this study through applying the proposed approach. When we analyze the results of the case study, it is evident that the proposed approach can considerably decrease not only time-to-market but also effort and cost for development. Moreover, the approach makes it possible to increase reliability and quality of software products, as well as the ease of maintainability. Therefore, the proposed approach is up-and-coming and utilizable in practice. Furthermore, the proposed approach and the tool developed can be applied to define a set of components for other domains.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this final chapter, conclusion and future work of the study are given. First, a brief conclusion of the thesis is presented. Finally, the future work related to this study is introduced.

6.1 Conclusion

In this thesis, a preliminary methodology is proposed to define a set of domain specific components in a product line. The main contribution of this thesis is to propose a systematic approach that contains a set of guidelines, activities, and metrics. In addition to that, the study presented the definition, planning, operation, analysis and interpretation of the case study that evaluated the viability of the domain design approach. Cloud monitoring domain has been selected as the case study in the scope of this thesis, because of its suitability for the validation of the proposed solution. Our current experimentation revealed the usability of this approach. The techniques were adequate in evaluating the determined set of components. Also, the suggested steps of the approach yielded a refined set that makes sense.

Furthermore, variability modeling is crucial to deal with variability and commonality in a domain. VCOSEML has enhanced COSEML for the inclusion of variability modeling at design level. Moreover, variability in both problem and solution spaces was made visible in the VCOSEML model. Thus, variability besides commonality is not only represented explicitly but is also managed through VCOSEML. Consequently, VCOSEML supports the capability to populate a product line environment

with domain specific components.

6.2 Future Work

As future work, the current version of VCOSECASE should be improved in various aspects. Firstly, the tool does not support representations of constraints appearing in extended feature models. Within this context, variability constraints and dependencies can be supported in the VCOSEML model. Secondly, a rule based system can be implemented for checking variability constraints. Thus, any invalid connectors among components can be detected automatically. On the other hand, the feature model from domain analysis is being mapped to abstractions manually. Feature models are currently being formed in the FeatureIDE tool. A feature model can be exported from the tool as an XML file. It is possible to import such a file by VCOSECASE. Thanks to the addition of the ability of importing feature models in XML format, the mapping phase of the approach will complete automatically. Thus, spent effort and time for completing this process will reduce.

Furthermore, the case study demonstrated the usability of this approach. Industrial scale case studies are missing until now. We are planning to improve the approach based on the experience to be gathered from its usage in real industrial projects. Our approach can further be refined as a result of applying to large domains preferably through real industrial projects.

REFERENCES

[1] T. Asikainen, T. Soininen, and T. Männistö. A Koala-based approach for modelling and deploying configurable software product families. *Software Product Family Engineering, LNCS 3014*, pages 225–249, 2004.

[2] M. Balci. Reference Architecture Tool for Software Product Line Engineering. Master's thesis, The University of Manchester, 2013.

[3] B. H. Barnes and T. B. Bollinger. Making reuse cost-effective. *IEEE Software*, 8(1):13–24, 1991.

[4] V. R. Basili, L. C. Briand, and W. L. W. Melo. INFLUENCES PRODUCTIVITY IN OBJECT- ORIENTED SYSTEMS REUSE. *Communications of the ACM*, 1996.

[5] G. Bayraktar. Representing Component Variability in Configuration Management. Master's thesis, Middle East Technical University, 2012.

[6] F. Belli. Dependability and software reuse - Coupling them by an industrial standard. In *Proceedings - 7th International Conference on Software Security and Reliability Companion, SERE-C 2013*, pages 145–154, 2013.

[7] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[8] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. 2000.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. 2001.

[10] S. Deelstra, M. Sinnema, J. Bosch, and J. Nijhuis. COVAMOF: A Framework for Modeling Variability in Software Product Families. *Software Product Lines*, pages 197–213, 2004.

[11] A. H. Dogru. Component-Oriented Software Engineering Modeling Language: COSEML. Technical report, Computer Engineering Department, Middle East Technical University, 1999.

[12] A. H. Dogru. *Component Oriented Software Engineering*. The Atlas Publications, Dallas, Texas, 2006.

[13] R. Dumke and A. Abran. *COSMIC Function points: theory and advanced practices*. 2011.

[14] B. Durak, E. Akbiyik, and I. Yigit. Deniz Savunma Sistemleri Alanında Sistematik Yazılım Yeniden Kullanım Yaklaşımı. *ceur-ws.org*, 1221:689–699, 2014.

[15] O. Eren. PL FSM : An Aproach and A Tool for The Application of Functional Size Measurement in Software Product Line Environments. Master's thesis, Middle East Technical University, 2014.

[16] C. Gencel. How to use COSMIC functional size in effort estimation models? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5338 LNCS, pages 196–207, 2008.

[17] C. Gencel and O. Demirors. Conceptual Differences Among Functional Size Measurement Methods. *First International Symposium on Empirical Software Engineering and Measurement ESEM 2007*, pages 305–313, 2007.

[18] L. Gherardi. Variability Modeling and Resolution in Component-based Robotics Systems. (February), 2013.

[19] M. Griss. Software Reuse: Objects and Frameworks are not Enough. *Object Magazine*, 1995.

[20] M. Griss. Software reuse architecture, process, and organization for business success. *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, 1997.

[21] M. Griss. Reuse Strategies CMM as a Framework for Adopting Systematic Reuse. *Object Magazine*, pages 60–62, 69, 1998.

[22] A. Guendouz and D. Bennouar. Component-Based Specification of Software Product Line Architecture. In *International Conference on Advanced Aspects of Software Engineering*, pages 2–4, 2014.

[23] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. Van Der Linden. Hierarchical variability modeling for software architectures. In *Proceedings - 15th International Software Product Line Conference, SPLC 2011*, pages 150–159, 2011.

[24] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers, 2004.

[25] I. Ileri, A. Eroglu, and A. H. Dogru. Component-Based Variability Modeling. In *The 18th International Conference on Transformative Science and Engineering, Business and Social Innovation*, pages 55–62, Campinas, São Paulo, Brazil, 2013.

[26] K. C. Kang, S. G. Cohen, J. a. Hess, W. E. Novak, and a. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, 1990.

[27] K. C. Kang and H. Lee. Variability Modeling. In *Systems and Software Variability Management*, pages 25–42. 2013.

[28] E. K. Karatas. An Ontology-Based Approach to Requirements Reuse Problem in Software Product Lines. Master's thesis, Middle East Technical University, 2012.

[29] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings - International Conference on Software Engineering*, pages 611–614, 2009.

[30] N. Kaur and a. Singh. Component Complexity Metrics: A Survey. *International Journal*, 3(6):1056–1061, 2013.

[31] G. Kotonya, S. Lock, and J. Mariani. Opportunistic reuse: Lessons from scrapheap software development. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5282 LNCS, pages 302–309, 2008.

[32] C. W. Krueger. Software reuse, 1992.

[33] S. Kumar, P. Tomar, R. Nagar, and S. Yadav. Coupling Metric to Measure the Complexity of Component Based Software through Interfaces. 4(4):157–162, 2014.

[34] A. Metzger and K. Pohl. Software product line engineering and variability management: achievements and challenges. *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 70–84, 2014.

[35] L. Northrop. SEI's software product line tenets. *IEEE Software*, 19(4), 2002.

[36] B. Ozyurt. Enforcing Connection-Related Constraints and Enhancement on Component Oriented Software Engineering CASE Tool. Master's thesis, Middle East Technical University, 2003.

[37] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*, volume 49. 2005.

[38] R. Prieto-Diaz. Status report. Software reusability. *IEEE Software*, 10(3):61–66, 1993.

[39] M. Razavian and R. Khosravi. Modeling variability in the component and connector view of architecture using UML. In *AICCSA 08 - 6th IEEE/ACS International Conference on Computer Systems and Applications*, pages 801–809, 2008.

[40] N. Salman. Complexity metrics as predictors of maintainability and integrability of software components. *Journal of Arts and Sciences*, pages 39–50, 2006.

[41] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[42] F. Van Der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. 2007.

[43] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. *Proceedings Working IEEE/IFIP Conference on Software Architecture*, 2001.

[44] R. van Ommering, F. van der Linder, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[45] K. Yadav and P. Tomar. Design of Metrics for Component-Based Software System at Design Level. *International Journal of Engineering and Technical Research*, 2(4):285–289, 2014.

[46] I. O. Yigit and A. H. Dogru. Yazılım Urun Hatlarinda Alana Ozgu Bilesenleri Belirleme Yaklasimi. In *Proceedings of the 9th Turkish National Software Engineering Symposium*, Izmir, Turkey, 2015. CEUR.

APPENDIX A

USER MANUAL

VCOSECASE is a modeling tool that provides graphical user interfaces for defining abstractions and components with variability among them. VCOSECASE is a new version of COSECASE in order to manage variability graphically. Therefore, COSECASE was improved to add new capabilities for supporting variability. Only variability capabilities are explained in this section. The newly added capabilities related variability provide to define and manage variability into a system. Following figures illustrates the steps how to define variability for an item of VCOSEML.

If a user wants to define variability for an element, the element should be selected and right clicking on it. Variability Management dialog is triggered when the "Variability" item on the pop-up menu is pressed. The pop-up menu and "Variability" item are shown in Figure A.1.

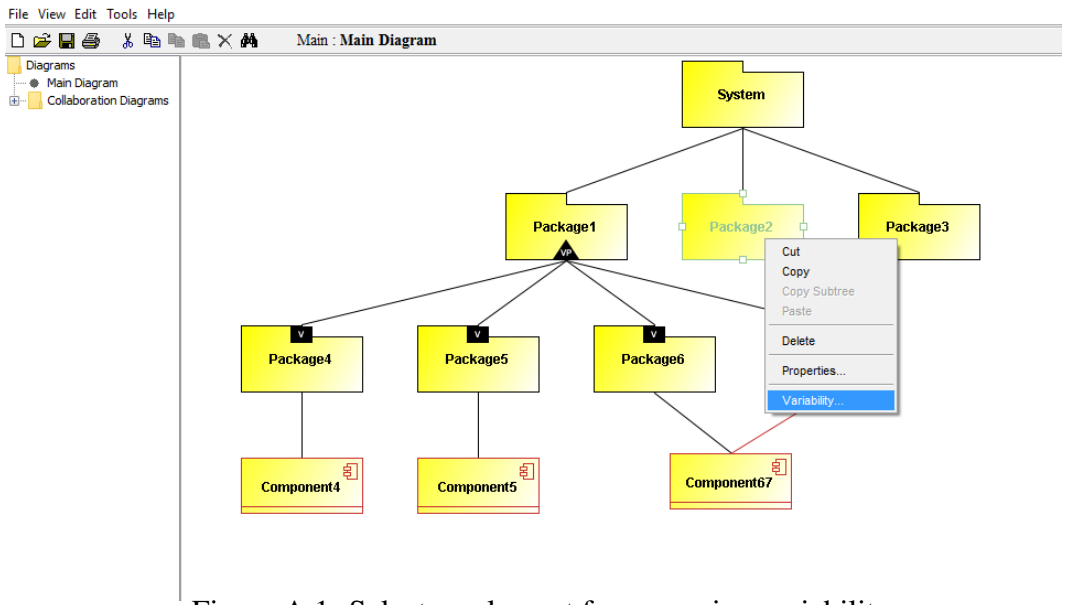

Figure A.1: Select an element for managing variability

Variation point and variant are two main terms in variability management as discuss in Chapter 3. The processes how to define variation point and variant are presented in Figure A.2 and A.3. When defined variation point for selected element, Binding Time property must be chosen. Also, the variability dependencies have to be selected in case of defining the variant of selected element.

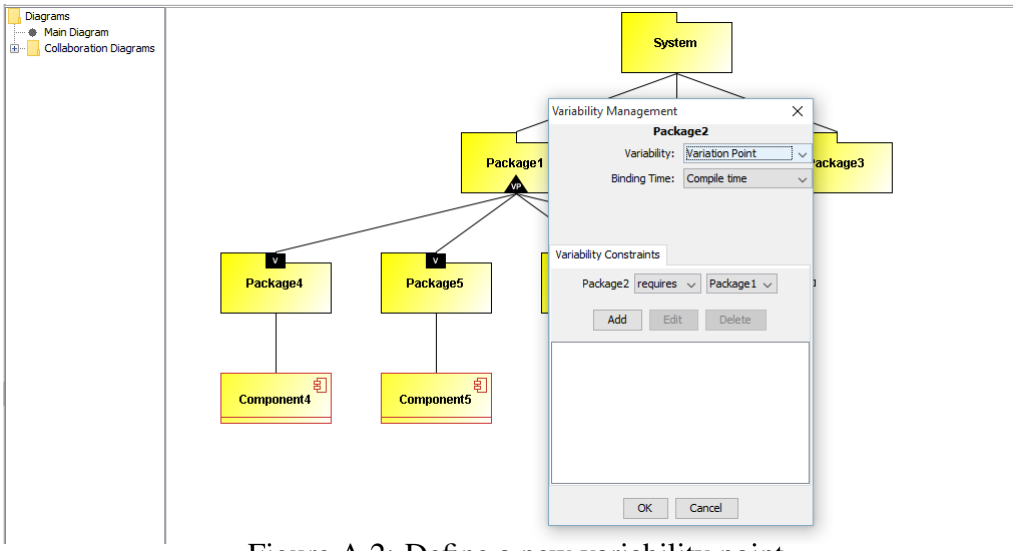

Figure A.2: Define a new variability point

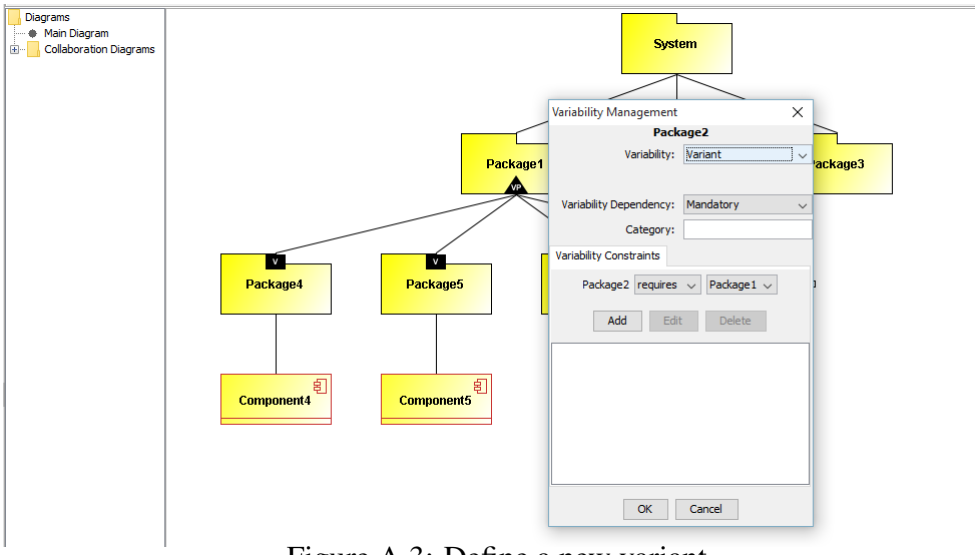

Figure A.3: Define a new variant

The step how to define requires and excludes variability constraints is illustrated in Figure A.4. A new constraint is added to the constraint list of selected element. All variability constraints of selected element can be edited or deleted from the constraint list.

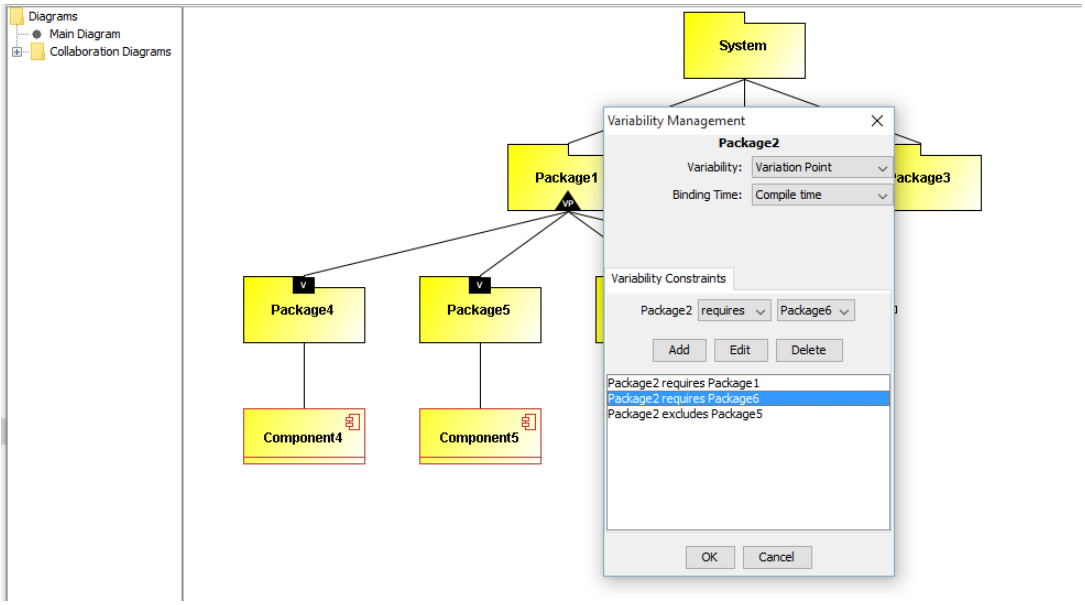

Figure A.4: Define variability constraints