REPRESENTING COMPONENT VARIABILITY IN CONFIGURATION MANAGEMENT

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

BY

GAMZE BAYRAKTAR

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

SEPTEMBER 2012

Approval of the thesis:

REPRESENTING COMPONENT VARIABILITY IN CONFIGURATION MANAGEMENT

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

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

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

Assoc. Prof. Halit Oğuztüzün _____
Supervisor, **Computer Engineering Department, METU**

Assoc. Prof. Ali Hikmet Doğru _____
Co-supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Assoc. Prof. Pınar Şenkul
Computer Engineering Dept., METU _____

Assoc. Prof. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Assoc. Prof. Ali Hikmet Doğru
Computer Engineering Dept., METU _____

Asst. Prof. Selim Temizer
Computer Engineering Dept., METU _____

Evren Çilden, M.Sc.
ASELSAN _____

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: GAMZE BAYRAKTAR

Signature :

ABSTRACT

REPRESENTING COMPONENT VARIABILITY IN CONFIGURATION MANAGEMENT

Bayraktar, Gamze

M.S., Department of Computer Engineering

Supervisor : Assoc. Prof. Halit Oğuztüzün

Co-Supervisor : Assoc. Prof. Ali Hikmet Doğru

September 2012, 61 pages

Reusability of assets within a family of products is the major goal of Software Product Line Engineering (SPLE), therefore managing variability is an important task in SPLs. Configuration management in the context of software product line engineering is more complicated than that in single systems engineering due to ”variability in space” in addition to ”variability in time” of core assets. In this study, a method for documenting variability in executable configuration items, namely components, is proposed by associating them with the Orthogonal Variability Model (OVM) which introduces variability as a separate model. The main aim is to trace variability in different configurations by explicitly documenting variability information for components. The links between OVM elements and components facilitate tool support for product derivation as the components matching the selected variations can be gathered by following the links. The proposed scheme is demonstrated on a case study about a radar GUI variability model.

Keywords: Software Product Lines, Configuration Management, Orthogonal Variability Model, Variability Management

ÖZ

KONFİGÜRASYON YÖNETİMİNDE BİLEŞEN DEĞİŞKENLİĞİNİN TEMSİL EDİLMESİ

Bayraktar, Gamze
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Halit Oğuztüzün
Ortak Tez Yöneticisi : Doç. Dr. Ali Hikmet Doğru

Eylül 2012, 61 sayfa

Yazılım ürün hattı yaklaşımının temelinde varlıkların yeniden kullanılabilirliği yatmaktadır, bu yüzden degişkenlik yönetimi yazılım ürün hatlarında en önemli ve kompleks görevlerden biridir. Konfigürasyon yönetimi (KY) yazılım ürün hattı çerçevesinde düşünüldüğünde geleneksel yazılım geliştirme süreçlerine göre farklılık göstermektedir, çünkü değişkenlik sadece zaman ekseninde değil, aynı zamanda özellik ekseninde de değerlendirilmektedir. Bu çalışmada, değişkenliğin çalıştırılabilir konfigürasyon birimlerinde, yani bileşenlerde, belgelenmesi me- todu ortaya konulmuştur. Bu metoda göre, değişkenliği ayrı bir modelde gösteren "Orthogonal Variability Modeling" (OVM) tekniği ile ilişkilendirilen bileşenlerde değişkenlik bilgisi açık olarak ifade edilir. Bu ikisi arasındaki ilişki kullanılarak seçilen değişkenliklere karşılık gelen bileşenler bulunarak otomatik ürün türetmeye yardımcı olacak bilgi elde edilir. Ortaya konulan yöntem radar kullanıcı arayüzü alanında bir çalışma ile örneklenmiştir.

Anahtar Kelimeler: Yazılım Ürün Ailesi, Konfigürasyon Yönetimi, Orthogonal Variability Modeling, Değişkenlik Yönetimi

to My Grandmother, My Family and Taylan

ACKNOWLEDGMENTS

I would like thank my advisor Halit Oğuztüzün, for his advice, criticism and supervision throughout this research. I also appreciate my colleagues for their guidance.

I would also express my deepest gratitude to my family for being supportive through all my life, and Taylan for his encouragement and support during this study.

TÜBİTAK, through the scholarship BİDEB, and ASELSAN, also supported this study, which I am thankful for.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

CDL:	Component Definition Language
CM:	Configuration Management
COVAMOF:	ConIPF Variability Modelling Framework
EMF:	Eclipse Modeling Framework
FORM:	Feature-Oriented Reuse Method
GEF:	Graphical Editing Framework
GMF:	Eclipse Graphical Modeling Framework
OSGi:	Open Services Gateway Initiative
OVM:	Orthogonal Variability Modeling
PLUSS:	Product Line Use case modeling for Systems and Software engineering
RADAR:	Radio Detection And Ranging
RSEB:	Reuse-Driven Software Engineering
SPL:	Software Product Line
XML:	Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Overview of Software Product Lines

In the emerging software market, companies are in a race to become the leader. In order to achieve this goal, the players try to launch more user friendly and higher quality software products on the market within very short time periods. To cope with the pace of such a market change, some of the most important factors that the companies have considered are decreasing the time to reach the market and the development costs, which led to a radical change, the so-called software product line (SPL). The product line concept arose from the automotive industry with Henry Ford, since then the concept became widely popular.

In the earlier stages of the software development, systems were analyzed, designed and developed one by one, without any notion of reusability. In the next step, reusability was only meant to copy codes among different software projects. However, nowadays the minds have been so changed that reuse of assets, documents, models etc. became a major issue. Reusability is very important because of its positive impacts on the cost, structure and quality of the products and time to market concerns of the companies. Software product line approach provides a platform for developing a family of software products in a specific domain rather than developing individual products separately. In a domain, products expose similar characteristics even though they include different features and properties. These similar characteristics might facilitate the task of analyzing the commonalities of the domain and developing software modules that realize one or more features of that domain. Such an approach requires an assets repository to be used for different software projects and products.

The main concept of the software product line can be stated as the domain. The needs of

this specific domain are identified as the first step of product line. After the analysis step, the main features of that domain are revealed. These features are combined according to the needs of a particular customer to generate a specific product. Differently from the single system development, in product line approach the selected features to be included in the final product are supposed to be developed as reusable assets beforehand. By properly configuring the assets which have already been developed, a customized product will be ready to market, and this is called mass customization [1].

Although the starting point of the product line concept was to produce same kind of products faster and in a more systematic manner, the demand for individually specialized products made the customization a must. If it is considered that different customers and/or companies might have different needs, the necessity of the customization will be better understood. As it might be guessed, customization means variability, therefore identification of both the commonalities and variabilities became one of the most important and critical issues.

Variability in software product lines is identified in the first phase of the process, which is called domain engineering. Not only the identification of variability, but also the asset realization is done during the engineering of the domain, that's why it is also called core asset development phase [2]. After domain engineering phase, application engineering follows. Variability, which was defined in previous step, is exploited in the application engineering process. In this phase, variability points are bound according to the features which are supposed to be in the finished product. Variability in software product lines defines the flexibility of SPL, therefore it is an important concern in these processes and is explained in the following chapter in detail.

1.2 Variability in Software Product Lines

SPL variability has significant differences from the one in the traditional software development process. In the traditional development approach, variation is generally managed in the product basis and in the time dimension. On the other hand, in software product lines managing variation over time axis is not enough. Instead, assets are configured according to the needs of a specific product, where these different configurations imply different versions of the asset, at the same time. This need creates another dimension, called ”variation

in space", which means different versions of a configuration item exist at a specific time [3]. Both managing variability in time and space creates a more complex configuration management (CM) problem. Traditional configuration management approaches deal with only time dimension, where variability in space should also be supported. Because of being such a complex problem, configuration management has gained importance and received special interest from researchers.

However, trying to manage the configuration in both time and space brings out different problems [3]. Because of complexity in variability management, several other methods have been suggested such as, feature modeling [4], COVAMOF [5] and Orthogonal Variability Modeling (OVM) [1, 6]. These methods are discussed in Chapter 2.

1.3 Motivation

In this study, configuration items, which are components in this case, are mapped to Orthogonal Variability Model, in order to manage the configurations of components using OVM, and also to map component information to variability model. Mapping variability information to different kinds of models is the basic aim of Orthogonal Variability Modeling. Although variability is modeled and mapped to various models, it is also important to observe the variability from the configuration management perspective. The components realize one or more functionalities and generally include variability which is modeled in OVM, that variability information is not directly available from a traditional CM system. As a result of this, when the components are used in a product, the variations for that product can not be provided which is valuable information to know. On the other hand, components are prone to change or they are configurable and different configurations or the changes generate new versions. In every version, variability information changes and these changes are not traceable, as the variability in space is not kept track of on a version basis.

The basic contribution of this work is providing "variability in space" in a traditional CM system by relating configuration items with OVM. The main purpose of this approach is to make variability information explicit for components by tagging them with this information in configuration management, in addition to version information. As a result, the variability information for different versions will become visible for the component and different ver-

sions can be compared according to their variability. Moreover, because the variability of the components used in a product is known, the overall variation of the product can be reported directly. Finally, new products can be derived based on reference architecture and selecting the variations needed for that product, which means supporting the application engineering process by using OVM-component relation.

1.4 Related Work

This study has similarities with the study of Ommering which introduced the Koala Component Model [7, 8]. This model also used components as basic configuration items and it put down methods that dealt with version management, variation and build support which are exactly what this study supports. In Koala, components provide interfaces and they are defined using so-called Component Definition Language (CDL). In this language, component's *requires* and *provides* interfaces. Although it also provides a method for supporting variability in configuration items, the method proposed in this thesis is more flexible than Koala approach. In that approach, the components should be modeled in Koala system, but in this study, there is no need for special language definition and variability modeled in OVM can be mapped to any kind of components.

In Brink and co-workers' study, a similar approach was followed, although the models taken into account was different [9]. In that study, a method was proposed to link features to different kinds of development artifacts. Feature models were used as a basis when creating links between features and other artifacts, which are requirements, implementation and test. With the help of these relations, the requirements and tests that correspond to a product configuration can be derived directly. The idea of associating features with artifacts resembles the relations between OVM and components. The main aim of Brink's study is to find the requirements or test artifacts belonging to a final product where thesis work aims to provide variability information. Moreover, using OVM hides the commonality details, therefore provides a more scalable view, whereas feature model includes commonalities, too, which results in a larger-scale model. Also, the integration with configuration management tools has not been referred in Brink's study [9]. However, the tool support for proposed method was mentioned as a future work.

In another study conducted by Heidenreich and colleagues, the main focus is that features are not enough to derive a product, because in feature models there is no information about how they are realized [11]. In order to derive a specific product, feature implementations needed to be known, therefore this information should be mapped to feature model. This main idea is similar to the Brink's and it is basically what is highlighted in this thesis. The term "variability mapping" was used also to describe this association.

In a recent study conducted by Anquetil and colleagues, existing tools and studies on traceability were reviewed and a framework for traceability throughout the software lifecycle was developed [12]. Although the traceability was considered from beginning to end, variability traceability was dealt specially. Traceability was separated into four dimensions, refinement, similarity, variability and versioning. Relating variability information to development artifacts which was proposed as a part of OVM specification [1], was referred as variability traceability. In that study, the importance of CM integration was also put forward.

In Roos's study [13], the quality attributes were related with OVM in order to support quality analysis. Although the purposes of these two studies are different, the approaches are similar which is worth to mention. Instead of quality attributes, here component information is related with OVM elements.

1.5 Thesis Organization

In the 2^{nd} Chapter, variation management in software product lines will be discussed. After that, Orthogonal Variability Modeling (OVM) will be explained in detail in Chapter 3 and then in the 4^{th} Chapter, the mapping of assets to variability model elements will be defined. The 5^{th} Chapter will introduce the OVM tool definition and design. In 6^{th} Chapter, a case study conducted in radar GUI domain will be explained. The usage of the tool will be explained in Appendix A.

CHAPTER 2

MANAGING VARIABILITY IN SOFTWARE PRODUCT LINES

In this chapter, the concepts related to this thesis will be explained. First of all, Software Product Line Engineering and its sub-processes will be introduced. After that, ”variability” and ”configuration management” terms will be examined and various variability management techniques from the literature will be reviewed.

2.1 Software Product Line Engineering

Clements and Northrop define the product line as ”a set of systems that share a common managed set of features satisfying the specific needs of a particular market segment.” [2] Although this is a general definition of the term, it exactly makes sense for software systems. In the definition, ”a common managed set of features” has to be identified by a pre-study, and after building the common set which is composed of core assets for software products' case, assets will be put together by using a systematic way in order to generate a new product. There are two main processes in software product line engineering, which are domain engineering and application engineering [1, 2, 14, 15, 16].

Product management deals with not only economic but also strategic aspects of a product. In this sub-process, the product scale for a specific domain, as well as the scheduling of the product release are created in accordance with the company vision. Identified product scale is used as an input for domain engineering phase and domain engineers identify the variability and commonality of the system from all aspects such as requirements, design, realization and test. The formal definition of domain engineering can be described as ”the process of software product line engineering in which the commonality and the variability of the product line are

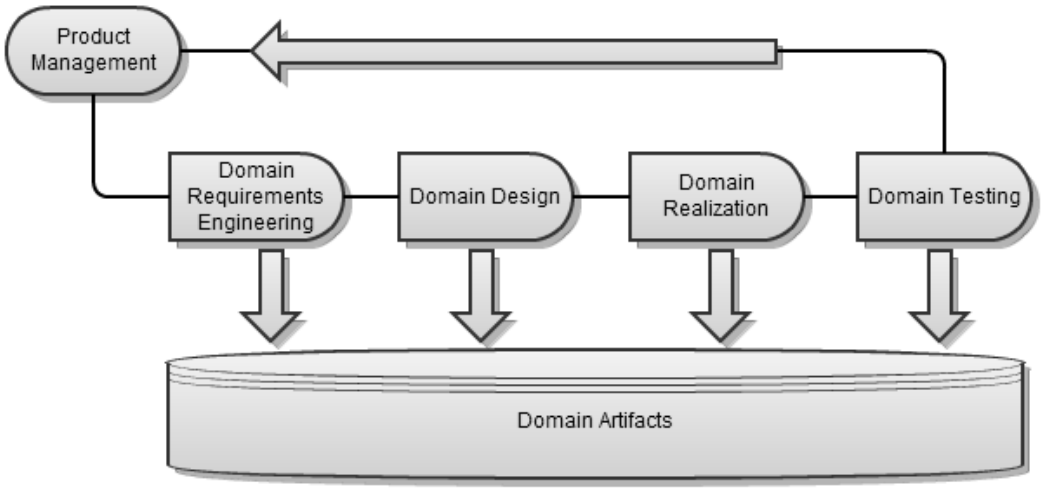


Figure 2.1: Domain Engineering (adapted from [1])

defined and realized." [1].

Different than single-system requirements engineering, domain requirements engineering step identifies a common set of functionalities that are shared by all products and that are specific to particular products. After requirement identification and reusable artifact specification, design phase follows. Design is a critical issue in domain engineering process, because it is the phase where a reference architecture is designed, considering variability points of the system. As the architecture is designed to be common for all products, the flexibility issue should be considered as much as possible for products which will be derived from this domain. In domain realization, the design is put in practice by applying low-cohesion and high-coupling principle. The assets which are realized at this stage of the process are generic enough to be reused in different products. They are also configurable to be customized for various products as a requirement of the mass customization. Reusable assets which are common for all products are called core assets. Domain testing is the final step of domain engineering. Reusable test artifacts are designed and developed to lower the time and effort, hence the cost of testing. The difference here is that assets realized in domain realization part are not really a part of a product yet, therefore there is no running application to be tested. Each asset is tested by using the assets in a simulation environment or individually. As a result, both test results and reusable test artifacts have been created. All of the output of sub-processes of domain engineering are called domain artifacts [1].

On the other hand, the second process is the application engineering which can be defined as ”The process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability.” [1] Application engineering is the process that ends with a finished product. This process is new to software product lines, that is, single system development has no notion of deriving a product, as only one product is generated from scratch at a time. Application engineering aims to develop and derive products by selecting certain features, reusing and configuring a base of already developed assets instead of writing code from scratch. First the products' requirements are identified, where domain requirements are used. As there is a reference architecture developed in domain engineering, the product's design is based on that structure. When the design phase is successfully over, product is derived by selecting assets mapped to the requirements of the product. If necessary, specific assets of a new product are created and also fed back to domain asset base. Moreover, already developed assets are configured according to the specific needs of that product and as an outcome, product is created with minimum sufficient analysis, design, coding and testing efforts [1, 2, 14, 15, 16].

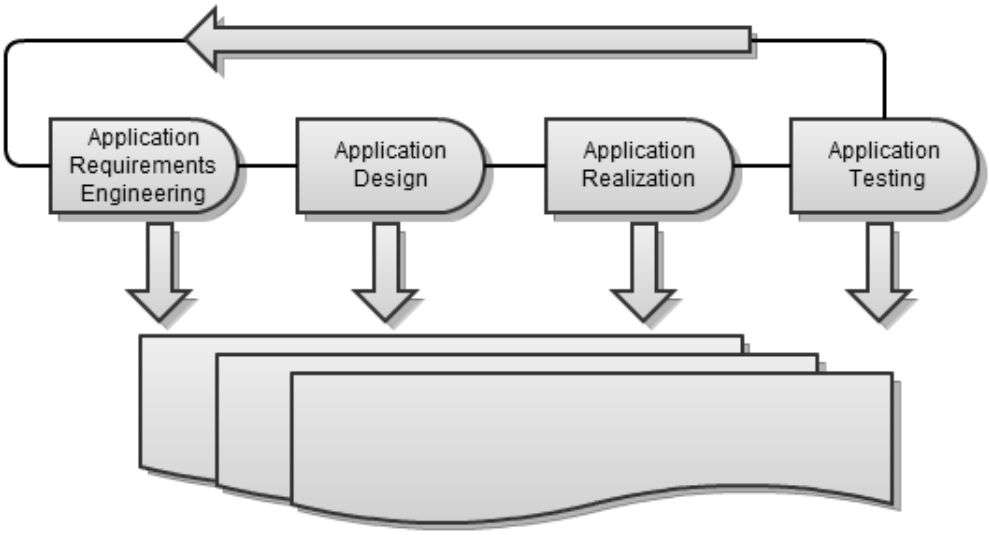


Figure 2.2: Application Engineering (adapted from [1])

2.2 Basic Terminology

In order to clarify the terminology, certain basic terms which are used in this study will be explained in this section. Generally speaking, requirements, codes, configurations, hardware and everything that different variations are wanted to be traced might be seen as a configuration item. In SPL, the focus is on the reusability issue. Reusable elements are specifically named as ”assets” and an asset offers a solution to a specific problem. An asset might contain one or more artifacts, which can be defined as basic configuration items [17]. As basic configuration unit, component is defined where the terms artifact and component are used interchangeably in Yu's paper [18]. An artifact/component might be a file, model or test case. They come together with usage specifications in a context to solve a problem, and make up an asset which might be variable or invariable. Variable assets provide a way to change the implementation or configuration for different needs. However, invariable assets are only used as-is [17].

There are two types of assets namely, core and custom (product specific) assets. Core assets are specific to a domain, but generic enough to be used in different products. Core assets form the skeleton of a domain and they generally state reference architecture to be used in all products of that domain. On the other hand, custom assets are specific to a product in the domain, and they are not necessarily reusable [18]. Core assets and custom assets are used together in order to create a product.

Although component term is used instead of the term artifact in [18], it is usually used in the same meaning with the term asset in many studies, such as [1, 2, 14]. The term component is used for both core and custom assets throughout this study. In order to keep the scalability of variability model manageable, components are used as configuration items in this study. That is, components are executable configuration items and basic building blocks of a product.

2.3 Configuration Management

Configuration management is the process of managing variability for both conventional software development and software product lines. The formal definition of CM by IEEE Computer Society is follows:

> a discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [19].

In fact, the aim of configuration management is simple, reducing the complexity rising from having many different versions of one configuration item, keeping track of changes as well as controlling changes among versions and being able to access a specific version at any time after creation [2]. Although the aim is simple, when the size of the systems is considered, it turns into a complex problem.

The basic requirements of configuration management activity are listed below [19]

1. The items that will be placed under configuration management are determined.
2. A configuration management system should be set up and maintained.
3. Specific versions of products should be created and released.
4. Change requests should be traceable.
5. Differences between product versions should be controlled.
6. Changes in interface definitions between software and hardware should be controlled.
7. Third party items should be managed.
8. Product releases should be managed.
9. In order to provide consistency between versions, configuration audits should be performed.

In traditional software development, configuration management is dealt from one aspect only which is ”variability in time”, whereas in SPL another aspect, ”variability in space” emerges. ”Variability in time” matches to classical configuration management. That is, the variation is managed in time dimension. However, ”variability in space” refers to managing different versions at a fixed point in time [1, 3, 20]. Actually Krueger [3] differentiates configuration management in traditional software development from the one in SPL. Although CM is used as a general term for both approaches, Krueger uses the term configuration management only for traditional case and variability management only for SPL [3].

The challenges specific to variation management in SPL are dealt in Yu and Ramaswamy's study [18]. It is very important to define the configuration items carefully, because the number of items that should be managed tends to be higher in product line approach than the single system development. In SPL, core and custom assets of domain are developed as individual configuration items, and their configurations should be managed separately. However, in single system engineering, configuration management is seen only from the product aspect, and the changes made to an asset do not affect the others. Using assets in many different products creates the problem of managing the changes of assets, especially the core ones. Because of the dependency between assets and products, the effect of a change in an asset might spread across different products and may become a complex problem. As mentioned earlier in this chapter, core assets are not only usable in all products of the domain, but also they are a part of the reference architecture. Therefore, detecting the shifts from using reference architecture or core assets is another challenge in CM for SPLs.

2.4 Variability Modeling

Since the software development activities become more and more complex, variability management has gained more interest and importance in software life development cycle. As a result, different modeling techniques have been developed. For both domain and application engineering processes, many studies have been conducted and different approaches have been suggested which are reviewed in [21, 22, 23, 24]. Some of the most popular techniques are feature modeling [4], COVAMOF [5] and Orthogonal Variability Modeling [1, 6].

Feature modeling approach which was proposed by Kang is one of the most popular approaches in variability modeling [4]. Feature, which means a collection of requirements or functionalities for a domain, is the basic unit of communication among different parties involved in development process, and used as an abstraction for distinct properties of a system [25]. This technique shows variation in feature diagrams whose nodes are the features. A node in the feature model might have one or more children showing differentiation for that parent node (feature).

Many studies based on feature modeling approach have been introduced [26, 27, 28]. FORM (Feature-Oriented Reuse Method) [26] uses feature models to design domain architectures and

components. In Reuse-Driven Software Engineering (RSEB) [29], use-cases are the basic factors for defining systematic reuse process. FeatuRSEB [27] associates the principles of feature modeling with RSEB. Similar to FeatuRSEB, PLUSS (Product Line Use case modeling for Systems and Software engineering) [28] method uses feature models to see the variability in use case models and aims at producing the models for specific products derived from family use case model.

Another technique for modeling variability is to extend use case diagrams to add variability information [30, 31, 32]. To make the variability information explicit in use case models and differentiate the types of variability, this information is integrated to use case diagrams.

Different than the methods explained up to this point, COVAMOF [5] and Orthogonal Variability Modeling [1, 6] are certain ways to model variability separately. Both approaches model variability in an external model which can be mapped to various models at different abstraction levels. COVAMOF provides a method to model variability in different abstraction layers which are features, architecture or implementation. Variability is modeled in these levels and variation in a layer can realize variability in a higher level. Moreover, different than OVM, dependencies are handled in a separate view, while they are shown as an integral part of OVM. The details of Orthogonal Variability Modeling will be explained in Chapter 3.

CHAPTER 3

ORTHOGONAL VARIABILITY MODEL

In this chapter, Orthogonal Variability Modeling and its basic elements will be explained and then an example OVM will be provided. Finally, rationale for using OVM in this study will be introduced.

3.1 Main Ideas in OVM

Orthogonal Variability Modeling [1] is a variability management method which deals with variability from different points of view. OVM was first defined by Bühne and his co-workers [6], and Pohl [1]. Their aim was to separate information of variability from other information in order to make its use more flexible. In this technique, variability is defined not in the actual software models like design diagrams or use case models, but instead a separate model is used for modeling variability. OVM defines the variability of software product line and also allows for different models like feature or component models to be related to variability information.

Bosch and colleagues address a number of general topics and issues about variability management in SPL that appear to be problematic in most variability modeling techniques [33]. First of all, the lack of *first-class representation* is explained. It is not possible to picture the variability information among different models. As there is no special way to represent variability, it is not possible to trace the dependencies among variabilities, so there is an obvious need for a standard way to represent variability which is available in every step during the development process. Besides the importance of availability of the variability information at all levels, it is also critical to see the dependency links between models and variability information which is mentioned as the second case. The other issue is the management of

variability information. Especially for large scale systems, the number of variability points and variants as well as their dependencies is so high that the management of this information is not quite easy. Tool support in such an environment is essential. The next problematic issue is the binding time selection of variation points. When the variability is exploited, it directly affects the flexibility of the system, and choosing the right time probably has a huge impact on the whole process [33].

OVM proposes solutions for those and also other cases which were mentioned in previous paragraph. First of all, OVM clearly offers a *first-class representation* for variability. Especially, this is the basic property of OVM; OVM represents variability information together with the relations between them in a separate model using a properly defined representation. The OVM definition is available at all levels. The dependency among other models and OVM is also proposed as a part of OVM specification. Moreover, OVM specification puts down explicit rules to be used as an input to the development process. As a result, a generated tool supports not only variability specification but also traceability links between different models and OVM. The last issue, which is the binding time selection, still continues to be a problem in OVM. There is not any explicit rule or information about binding time in OVM.

The variability models, which are integrated to other models have certain disadvantages over modeling variability in a separate model which are [1, 6]:

1. consistency between models,
2. traceability of variability information across models,
3. increase in complexity with variability information,
4. different variability concepts of different models,
5. ambiguity in development artifacts.

Orthogonal variability model is a way to overcome these disadvantages. In OVM, variability is defined in a separate model which can be related to other models like requirements, design or test models. OVM defines variability of a software product line by using basic elements which will be explained with their graphical views, in Section 3.2.

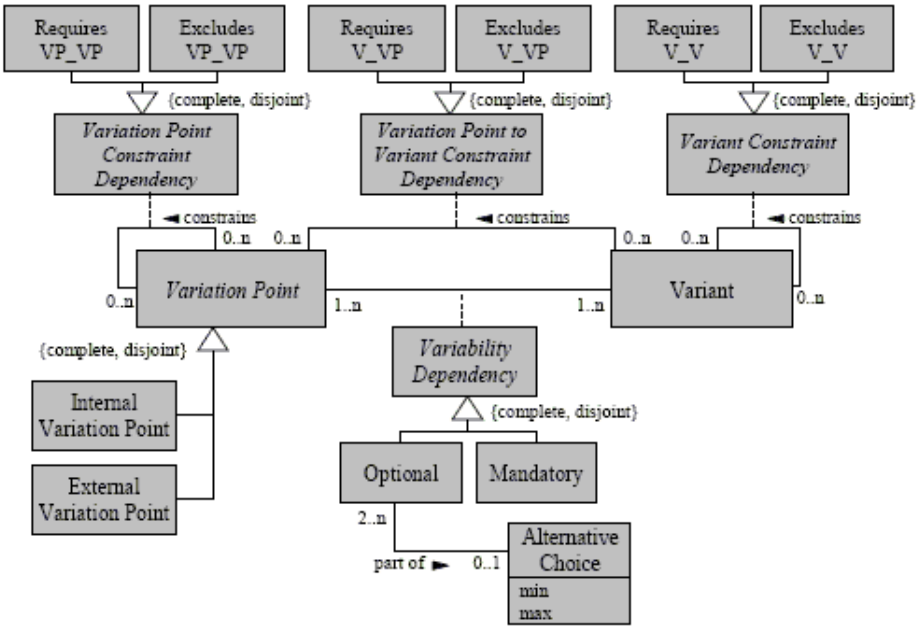


Figure 3.1: Conceptual Model of OVM (adapted from [1])

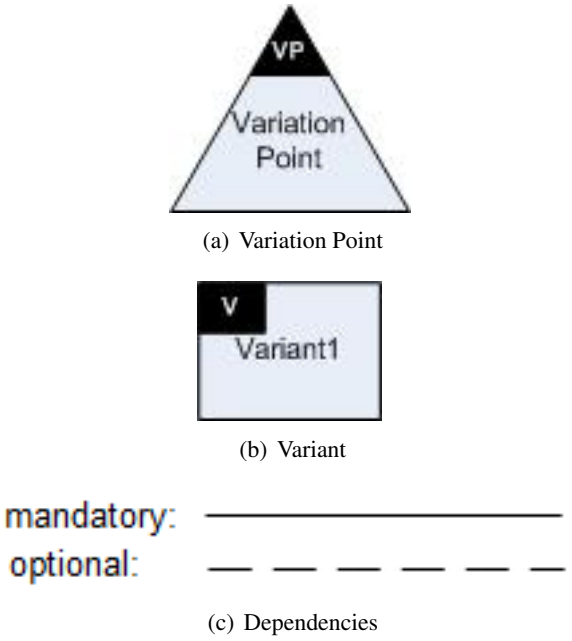
3.2 OVM Elements

In order to define the variability in a separate model, a meta-model is specified by using basic elements. Variation point and variant are two basic elements that are used to create an OVM. OVM elements and their graphical representations are proposed by Bühne et al. [6] and Pohl [1]. Throughout this study, variation point and variant are together named as OVM elements for the sake of simplicity, and also in many cases they show similar properties.

Figure 3.1 shows the basic elements of OVM and the relations between them. In the following paragraphs, they will be examined in detail.

Variation points can be defined as the differentiation spots in a software product line and they also show and highlight the reason of the presence of a variation in SPL [1, 6]. The graphical notation of a variation point is demonstrated in Figure 3.2(a).

Variant matches to the possible different elements for a variation point. The term variant was also mentioned by Bosch [14] exactly in the same manner; however it was not in a special context. Figure 3.2(b) shows the graphical view of a variant.



(a) Variation Point

(b) Variant

(c) Dependencies

Figure 3.2: OVM Elements

Variation point and variant have a parent child relationship called Variability Dependency, which might be in two different formats: mandatory and optional (Figure 3.2(c)). As their names suggest, mandatory variability dependency is used for the variants that should be included if and only if its parent variation point is selected; while optional variability dependency is for the variants which might or might not be a part of the selection. Another dependency type comes with an optional dependency which is alternative relation. This type of dependency relates a group of optional variants and shows the amount of variants that might be selected.

Constraint dependency is another relationship type between variation points and variants. Constraint dependencies are similar to cross-tree relationships in feature diagrams. Two types of constraints are ”requires” and ”excludes” constraints. Requires constraint defines the requirement among OVM elements. That is, if an OVM element A requires OVM element B, when the OVM element A is selected, then B should also be selected. On the other hand, excludes constraint defines just the opposite. If OVM element A excludes OVM element B, then A is selectable if and only if B is not.

The constraint dependency is specialized according to the types of OVM elements that are involved in the constraint. They are Variation Point-Variation Point, Variation Point-Variant

and Variant-Variant constraint dependencies. These types are valid for both requires and excludes constraints. In the scope of this study, these specialized constraint dependencies are not used, only bare excludes and requires dependencies are included, because special forms do not bring new features to the constraints.

3.2.1 Example OVM

Figure 3.3 shows an illustrative example that belong to the radar domain. In Chapter 6, radar domain will be introduced and explained in depth.

A small model is shown in Figure 3.3 to give an insight about both a radar scope and an orthogonal variability model. Basic elements of OVM are used in this model. Here, *RADAROutput_Track* is a mandatory variant of *RADAROutput* variation point, where *RADAROutput_Plot* is optional. Variation point *Sector* requires variant *Scope_Scope1*. It can be seen that the three variants under *Scope* variation point are alternative relations.

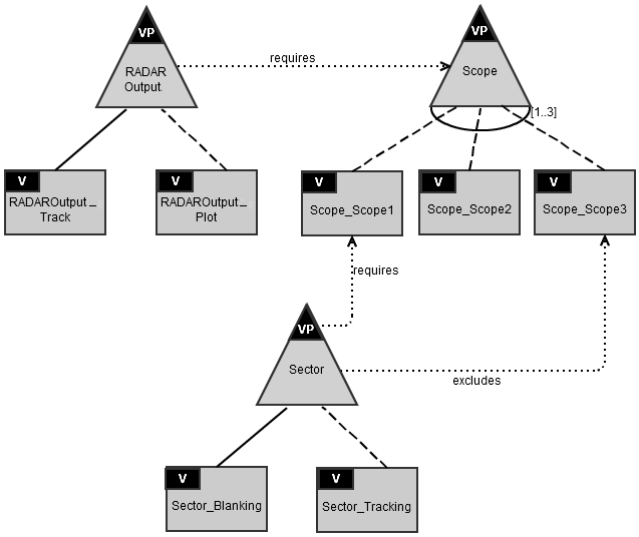


Figure 3.3: Example OVM

3.3 Variability in Different Kinds of Artifacts

As mentioned before, Orthogonal Variability Model defines variability in a separate model that can be linked to any other models at any stages of the development process, which is the

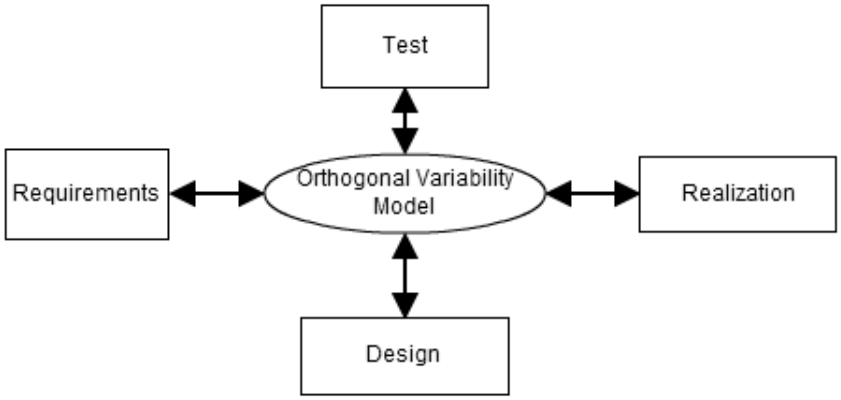


Figure 3.4: OVM and Other Stages of Development

basic advantage of orthogonality (Figure 3.4).

In order to show the dependencies among other models, artifact dependency is defined. The artifact mentioned here can be anything in the development process. Both variation points and variants can be associated with the development of artifacts. The development artifact-variation point/variant and variation point-development artifact relations are 0-to-n; whereas the variant to development artifact relation is 1-to-n. That is, a variant should definitely map to at least one development artifact [1].

3.4 Reasons for Using OVM

Various modeling techniques have been introduced to represent variability and relations between elements some of which are explained in Chapter 2. These models focus on various different aspects of system. They also include functionality common to all products in the model, together with their interrelations. This adds an unwanted complexity for the model of the system, because most of the time the commonalities are not the case to be considered, but the variabilities are. In order to scale the model to a more feasible form, the commonalities might be discarded from the model. Another point is modeling variability integrated to other kind of models like design, implementation or test. In order to use the variability information outside these models, first it should be extracted somehow, which might turn out to be a complex task. Rather than that, models that just focus on variability of the system are preferable. OVM and COVAMOF are such kinds that model the variabilities of a system separately. CO-

VAMOF uses two different views, one for variability and the other for dependencies between the variations. In this technique, the variability is modeled in different abstraction levels and variation points can be realized by the variation points in a higher abstraction level. Having different abstraction levels and using a separate view for dependencies make COVAMOF more complex than OVM. OVM technique is the one that provides variability in a single model together with dependency information which provides a compact view for overall variability in the system at one glance. Therefore, it is easily adaptable for linking variations with different parameters.

As mentioned previously, Roos and co-workers conducted a study that uses OVM for quality based analysis [13]. In this study, quality parameters were linked to OVM elements, and by using this information, automatic analyses were conducted. That work showed that linking attributes to OVM and effective use of the linked information is possible.

CHAPTER 4

RELATING OVM AND COMPONENTS

In this chapter, first, representing variability information in configuration management will be explained, and then the details of the association between OVM and components will be clarified. Next, how the version, and build and release management issues are handled will be defined. Finally, the chapter will be concluded with a discussion of the advantages and tool support for the proposed approach.

4.1 Expressing Variability in Configuration Items

As explained in Chapter 3, OVM defines variability information in a separate model, and the artifacts produced during the software development life cycle can be annotated with the elements of this model. During the domain engineering phase, after requirement analysis and domain design, products are realized by composing core assets of that domain. Besides core assets, for specific needs of different products, custom assets are also developed. They are both called as development artifact or component. Explicitly defining the relations among several components determines the component model which shows the dependency, realization and generalization relations.

In this study, a new component-OVM relation is defined. Although the association is between components and OVM, no component model is assumed, but the components themselves are used, different than Pohl's approach [1]. There are two main reasons for not using component model. First of all, the component model is static, which means it defines structural relations among components and interfaces. Besides that, the importance of variability in time and space is emphasized, however no version information is mentioned. Variability in component

models can be expressed through OVM, however the lack of version makes it impossible to compare variability between versions, which stands as the second and most important deficiency of component model used by Pohl. There is no notion of version for components which is important information, as during the development many different versions are created which might map to different configurations or implement different functionalities. The main aim of this study is to make variability traceable, therefore version not the relations between components, is the focal point. Although these component relations should be known for other needs which are explained in Section 4.5, for the traceability of variability information, these dependencies are out of scope. That is why component model does not fulfill the requirements in this case.

In this new mapping, components are seen from the configuration management perspective, where they are individual configuration items with version information. The interdependencies between components are not of interest in this view but the versions, therefore each component is shown separately. As they have already had versions supplied by configuration management, variability information for components/configuration items can be inserted and kept in configuration management. The main aim of configuration management is to trace the differences among versions, and traceability of variation is also a valuable knowledge to be kept. In order to keep variability information in configuration items, it is introduced as a new tag, similar to version, in the configuration management system.

The goal of expressing variability information in configuration management is facilitated by using an already existing attribute and introducing new attributes for components. They are explained below.

Version (existing): specifies version information for the component.

OVM Element List (new): specifies the OVM elements that this component realizes.

Status (new-derived): specifies open/close status of the component. If a component is open, it contains at least one variation point, which is unbound. If there is no unbound variation point, then it means that the component is closed. It shows summary information to the application engineer about whether there is any unbound variability for selected component or not. This attribute could be derived by using OVM element list attribute.

Version attribute is nothing but the version given when a change is made to the component

or it is configured according to the specific needs. Actually this information has already been supported by all CM tools. Status information might be inferred by inspecting the attribute OVM element list or it might be set manually. The most important attribute that is newly introduced by this study is OVM element list which is the variability annotation mentioned previously.

4.2 OVM-Component Association

From the component perspective, adding variability information as a new attribute to component described in previous section, means also relating OVM with that component because the variability information is gathered from OVM of the domain. From the OVM side, the components related with that OVM element is also valuable information, in order to find those components realizing the variant or holding the variation point.

The importance of binding time information for variation points is addressed in Bosch's study [33] as binding time selection. If variation points are bound earlier than required, the flexibility of product line decreases, and if bound later, the cost of development increases. Because binding time directly affects the flexibility of a product line, it is important to make this information explicit in variability model by introducing attributes for OVM elements. The following three attributes belong to OVM elements.

Binding Time: specifies when to bind the variation point.

Binding Method: specifies how to bind the variation point.

Component List: specifies the components realizing the OVM element. This is also a 1-to-n relation.

To be more specific, binding time and binding method are only applicable for variation points as it might be guessed; and the last attribute is for both variation points and variants. The importance of binding time selection is mentioned and highlighted in many studies [1, 33, 34]. The binding times might be listed as; before or at compile time, at link time, at load time and at run-time [1]. Besides deciding the right time for binding variability, it is also very important to know how to bind it. This information is saved in binding method property. By looking at this property, the correct method can be applied during binding process. The last property

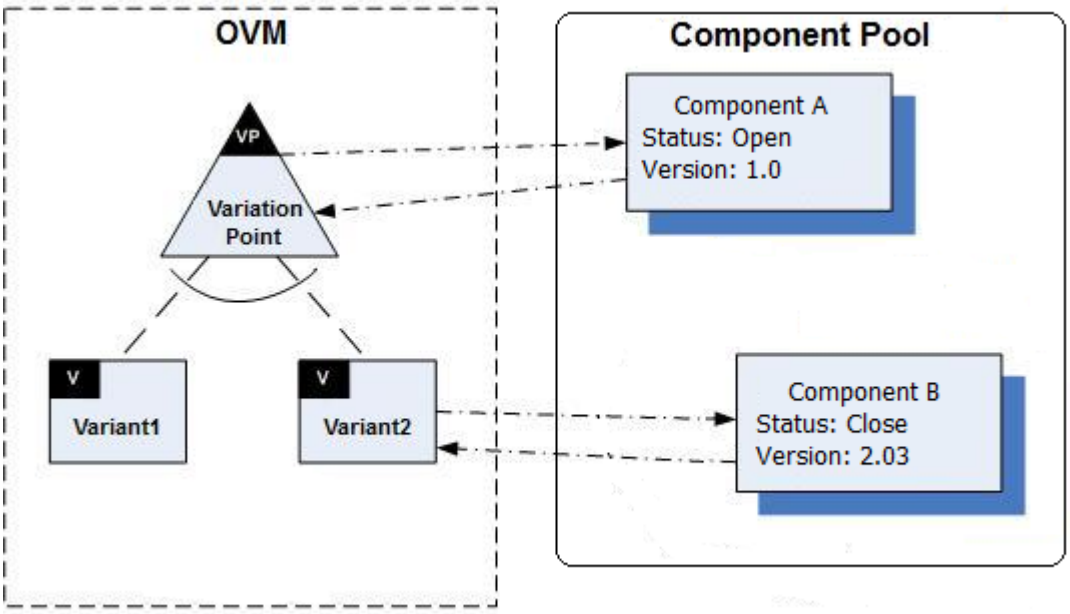


Figure 4.1: OVM and Component Mapping

represents the component list that the OVM element is realized by. When that OVM element is selected to be included in the final product, by using the component list, the components that should be a part of product are easily determined.

As seen in Figure 4.1, there are links in both directions between OVM and components. It is necessary for two things; first of which is that after selecting variations to be included in final product, the corresponding components should be identified using the links. The second issue is tracing the variation points and variants that the component realizes. The former is explained in OVM as development artifact dependency [1], however latter is a new concept proposed by this study. Both links are transferable to each other. To be more specific, if one kind of link is explicit, the other direction might be inferred. Moreover, the notation used in OVM-component links is the one proposed by Pohl [1], which is explained in Chapter 3.

In Figure 4.1, components are not drawn as a component model; but they are shown independent from each other, that explains why it is named as ”Component Pool”. Although the dependencies among components are not shown in Figure 4.1 explicitly, they are available (e.g. through manifest files for Java components). These dependencies will be used in product derivation. The scenario for creation of OVM-component relations and product generation;

and what manifest files are, will be explained in Section 4.5.

4.2.1 Basic Assumptions

Before going through the details of the study, the assumptions made during the study are listed, which are essential to fill in the gaps and prevent the inconsistencies that might arise.

- OVM model is supposed to be well-formed.
- The names of variation points are unique for a model. The uniqueness is important, because reporting and analyzing operations on OVM are done based on that assumption and OVM elements are searched by their names.
- OVM modelers might assign the same name to different variants, whose parents are different variation points. In order to guarantee the uniqueness of variants, the following naming convention is imposed. This convention also helps to see the variation point that the variant belongs to at first glance: *"Variation Point Name"_"Variant Name"*
- Component dependencies of each component are available. This dependency information might be gathered from the component itself, which might be embedded in different formats depending on the type of component. (For example, Java components, which are *jar* files, have *manifest* files that include this dependency information. Details of manifest files and how the dependency information is extracted are explained in Chapter 6.)

4.3 Creation of OVM-Component Relations

The first step of creating the relations proposed here is to create the OVM for that domain as a part of domain engineering process. The process of OVM creation defined clearly by Pohl [1], therefore the details will not be mentioned here. However, the attributes defined for OVM elements should also be specified. OVM creation is performed during the first step in the domain analysis process, therefore there are no components present yet. In domain realization, the components are determined and implemented, and the information about which component realizes which OVM element is known implicitly by the developers. By using the approach proposed, this implicit information is saved in the OVM element binding attribute

of the component.

There might be different methods for both creating OVM and OVM-component association model. One of these methods is to use a configuration management tool. All configuration management tools support adding new tags to configuration items, so variability information can be introduced as a new tag to components, like version information, and it can easily be kept under configuration management. If no configuration management tools are used, this information can be saved textually in a separate document.

The approach proposed in this study is easy and flexible to be implemented in different platforms. The attributes defined previously for components can be adapted to different configuration management tools, because there is no special requirement besides adding variability attributes. The best method is to use a tool integrating both OVM and configuration management by supporting the definition of OVM elements and components with the attributes mentioned. The discussion about a tool support is given in Section 4.7.

4.4 Version Management

The components, developed to realize a functionality are saved in a repository and there might be different versions of a component in this repository. One of the basic properties of a Configuration Management tool is to support adding version tags to components in order to track the changes among versions. In this study, it is proposed that, adding variability information to components provides a way to track the variability between versions.

If a new version is added to component pool, this new version should also be defined together with OVM annotation. The new version of the component might have different OVM element relations than the previous version, for example previous version might contain a variation point which is unbound and in the new version this variation point might be bound to a variant. This new binding relation is also defined for that version and saved in the OVM element binding attribute of the component. By looking at that attribute, the variability differences between versions are easily analyzed and reported.

In order to clarify the proposal in the previous paragraph, consider the following example; component A having version 1.0 realizes variation point VP1 and variant VP2_V1, so the

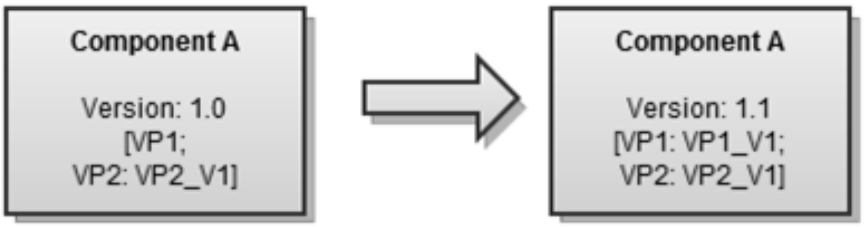


Figure 4.2: Version Management

OVM element binding attribute's value is [VP1; VP2: VP2_V1;]. A change is made to component A and its new version is created as version 1.1. In this new version, the variation point VP1, which is unbound in version 1.0, is bound with the variant VP1_V1, therefore the value of OVM element binding attribute changes to [VP1: VP1_V1; VP2: VP2_V1;]. If the 1.1 and 1.0 versions are compared, it will be seen obviously that the variation point VP1 is bound to VP1_V1 (Figure 4.2).

4.5 Build and Release Management

In application engineering, the aim is to exploit variability created in domain engineering and to create the final product. To reach this goal, the OVM-component relations as well as component dependencies should be used effectively.

First of all, there should be a feature model from where application engineer selects the features of the product to initiate the process. The feature selection and creation of OVM for the specific domain is out of the scope of this study, but that process is assumed to be done already. Also, core components that are common to all products are assumed to be included. The focus is on creating the relations between OVM and components, which pictures the variability of system and produces the final result.

In creating the relations, there are two ways: create the relation from OVM to component if the component realizing the OVM element is known and already present in the component pool or create it while adding a brand new component or a new version of an existing component. The more natural way is the second one, because the time of OVM creation is supposed to be the domain analysis part and there might be no components developed in the time of OVM creation. On the other hand, when the component is developed, the variability realized by the

component is already known. As mentioned earlier, it is enough to create the relations in one direction, from components to OVM elements, as the other way might be derived.

Now OVM, components and their relations are ready, it is time to move on to the next step, which is the product generation. In order to derive a new product, there should be reference architecture to be based on. As expressed previously, the common assets provide this reference architecture and as the core or invariable part of the domain is out of the scope of this study, they are assumed to be selected on the basis of the necessity of the presence in the final product. Our focus is only the variability of the system. The process of product derivation is explained below.

- Application engineer selects the variation points or variants that should be in the product.
- By following the links of OVM elements, corresponding components are found.
- The component dependencies of found components are extracted. This information is integral to components and not visible in the model, however these dependencies are so important that without a dependent component the product would not be complete. Therefore in the post processing step, by extracting the dependencies, required components are selected automatically if they are not already added as a result of the previous process.
- After completion of adding depended components, all of the components of the final product will be present to compose the product.

4.6 Advantages

Making variability information explicit together with version information for components brings certain advantages from two different aspects, first is OVM elements and second is components.

First of all, in this study, attributes are added to OVM elements for making binding information visible in OVM. As explained before, binding time is critical information for variability. To specify the correct time for binding of variability has a great impact on the variability of the system. Secondly, proposal made here realizes the approach proposed by Pohl [1] which

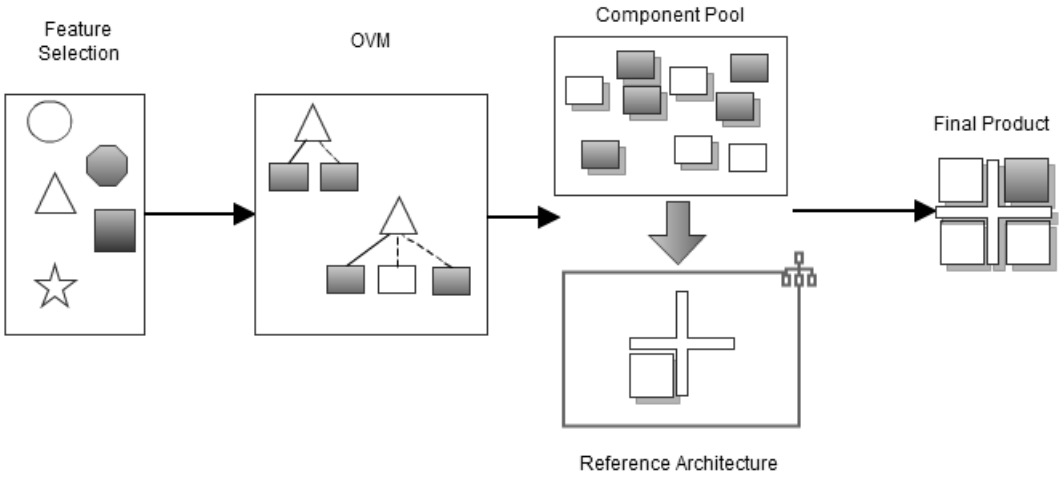


Figure 4.3: The Product Derivation Process

is to map OVM to development artifacts. It is realized in parameter ”component list” defined for OVM elements. The advantage of having such a list is automatizing the operation of component selection, after selection of OVM elements. By looking at the component list that the OVM element is realized by, the components can be found from the component pool.

Second aspect of the study is about the components. The components' parameters are version, status and OVM elements that are realized by the components. Version information for a component is essential since both variation in time and space are reflected by subsequent versions of components, and also historical background of the component can be tracked using version information. Status parameter gives an insight about the status of the component, and the user understands whether there is any unbound variability or not. The last one is having OVM element list and it is probably the most important parameter proposed by this study. This list can be used to analyze variability information for not only the component but also the final product. By using the component list information in OVM elements and OVM element list information in components, different analyses may be conducted. The analyses might be specific to a product or general to the domain, such as the following to be

- components with their versions and corresponding OVM elements included in the product.
- differences among component's various versions in terms of variability.
- unbound variation points in a component.

- bound variation points and which variants are used to bound.
- the component list realizing the variation point/variant.
- the OVM element list that the component realizes.
- any inconsistencies among OVM elements and also components that might arise from dependencies in OVM.

4.7 Adaptation for Tool Support

The components which belong to a domain are usually saved in a configuration management (CM) tool. Configuration management tools allow saving different versions of components separately, and retrieving specific versions or comparing versions. These tools make life easier in software development and there are many offerings, such as SVN [35], CVS [36], GIT [37], and ClearCase [38].

The proposed parameters for components' version, status and OVM element that are realized by this component can be saved easily in any configuration management tool. All configuration management tools support versioning of components; therefore the first parameter is already a part of every CM tool. Second and third parameters can easily be integrated to CM tools either as tags or as special comments as every CM tool supports adding special information about configuration items.

The method defined here, which is adding binding time, method and component information to OVM elements are, flexible to be added to any tools supporting OVM definition. The proposal made here is easily applicable to any OVM tools. The only thing is to add attributes to a variation point to specify binding time and binding method. Binding method simplifies the work of application engineer by specifying how to bind the component. This attribute can be specified in many different ways. For example, if the variation point is bound according to a compile flag, it can be specified as listing the flags that can be used. The process explained up to here is realized as a case study which will be presented in Chapter 6.

CHAPTER 5

TOOL SUPPORT

In this chapter, first, the tool requirements supporting the defined rules, will be clarified. Then, the design details together with the environment used to realize the tool will be explained.

5.1 Requirements

In order to accomplish a good design, the requirements should be specified precisely, which makes requirement analysis an important activity. Theoretical definition of OVM has already been put forth by Pohl clearly and also component OVM relation has been defined in Chapter 4. These definitions form a basis for stating the functional requirements for providing tool support. The functional requirements define the scope of the tool and they are derived from basic OVM definitions as mentioned above. Besides the rules helping to define functional requirements, also non-functional requirements should be identified, as they determine the quality, usability and maintainability of the application.

Both functional and non-functional requirements are proposed below.

- The tool should provide a graphical user interface to define an OVM using the OVM elements and also the dependency connections, which are variation point and variant elements, constraint dependencies, optional, mandatory and alternative variability dependencies and artifact dependencies. The graphical representations of all elements and dependencies should be in accordance with Pohl's definition.
- There should also be support for defining component and OVM element-component relation graphically.
- The OVM elements and components should be named uniquely when created.

- The defined model should be saved and restored after creation. The tool should save the model using a known file format, namely XML.
- The tool should support add, edit and delete operations for all items that are introduced for the model.
- The tool should support attributes for OVM elements and components. These attributes were introduced in Chapter 4, which are binding time, binding method and component bindings for OVM elements; version and OVM element bindings for components.
- The tool should allow hide/show dependency connections on the graph.
- The tool should provide toolbar/menu listing the elements that can be created, and also explanatory icons for the elements in the list.
- The tool should prevent adding invalid dependencies. For example, mandatory dependency can be created between variation point and variant, not between variant and variant.
- The tool should support moving the elements in graphical definition, when the element is dragged with mouse.
- The tool should support adding dependencies by selecting the source and destination on the graph.
- The attributes of an element should be examined in a separate window.
- The tool should give feedback to the user in case of an error, stating the cause of error.
- The tool should also support the analysis operations on OVM. The analysis operations that should be supported are
  - variation points and variants those are represented/realized by a component,
  - components that represent a variation point,
  - components that realize a variant.

These requirements constitute the basis for a tool that supports both OVM definition and component relation introduced as a new aspect. By using these basic requirements, a tool can be implemented in any programming language. Java is used in this thesis.

5.2 Design Details

Pohl [1] states a predefined way to represent variability model, with the elements and their graphical representations. Not only graphical representation but also relations and rules are defined clearly. The way Pohl stated is realized in Eclipse Graphical Modeling Framework (GMF). In addition to OVM, components are also augmented with the properties specified in Chapter 4. First of all, Eclipse GMF is explained in the following section in order to make the design details perfectly clear.

5.2.1 Eclipse GMF

Although Eclipse generally refers to the development environment that supports different programming languages such as Java and C/C++, it also provides a common platform for different applications. Using Eclipse API, different development tools have been implemented on top of this common platform. Eclipse Graphical Modeling Framework (GMF) [39], Eclipse Modeling Framework (EMF) [40] and Graphical Editing Framework (GEF) [41] are some examples to those kinds of applications.

Eclipse EMF provides a framework to specify a model of specific domain. It not only enables the user to define the model but it also generates the corresponding Java classes and these classes are used for both viewing and editing functions supported by graphical editors. Eclipse GEF is one of these editors and it supports drawing models using specified symbology. Eclipse GMF is a framework that integrates EMF and GEF into an infrastructure that enables the definition of specific graphical editors for specific domains. Using this framework, first the model, and then its graphical representation are defined.

The general process of a graphical editor definition for a domain model is shown in Figure 5.1. First of all, domain model should be introduced as GMF's *Ecore* model. GMF provides a graphical user interface (GUI) for domain model definition, where domain elements, their relations and attributes are defined as a class diagram. After domain model, the graphical part of the editor is specified. There are two parts of graphical definition; graphical and tooling definition. The former refers to the graphical view of elements defined in *Ecore* model, that is to say, the symbology of the editor. The latter is the specification of the tools used to create the elements or links of the diagram. It can be referred as a menu or a toolbar of an application.

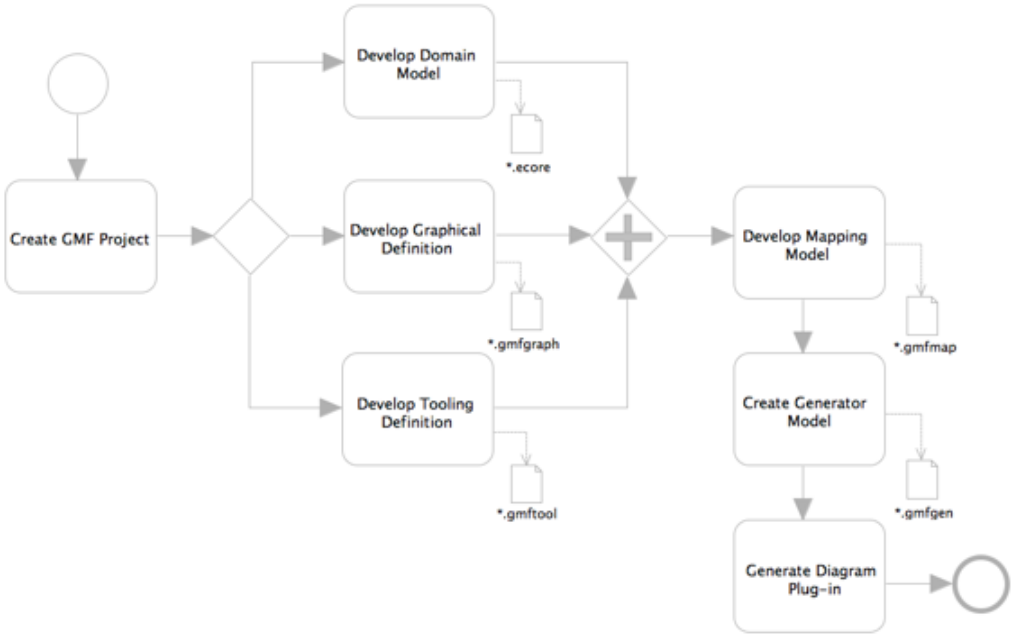


Figure 5.1: Eclipse GMF Process Overview [39]

Up to this point, everything related to domain and graphical representation has been defined separately, and now they must be related to each other so that they have a consistent meaning. This mapping is done automatically when *Ecore*, graphical and tooling models are specified, however it is also possible to change it manually. In the final step, the mapping model is converted to generator model from which the code for graphical editor application is generated. As a result, an application specific to the domain model and the diagram is created.

Eclipse GMF provides a framework and user interfaces for all steps, therefore it is easy to create a new application without writing a single line of code. The hardest part of using GMF is to understand the idea under the hood which has been explained in the previous paragraphs. After the idea is grasped, the usage of GMF is straightforward. Using graphical interfaces makes the definition of the models easy, and abstracts the code details which stand as the main advantages of GMF.

5.2.2 OVM Tool Design

In order to start to build a graphical editor, first a GMF project should be created. After that, Eclipse GMF provides a dashboard to guide the user during the whole process (Figure 5.2).

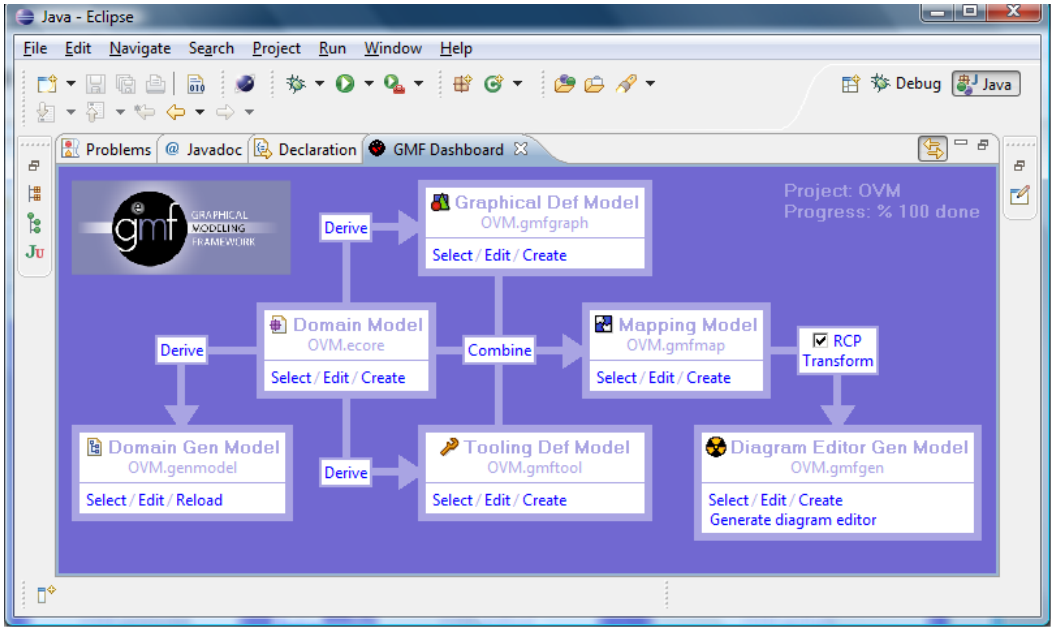
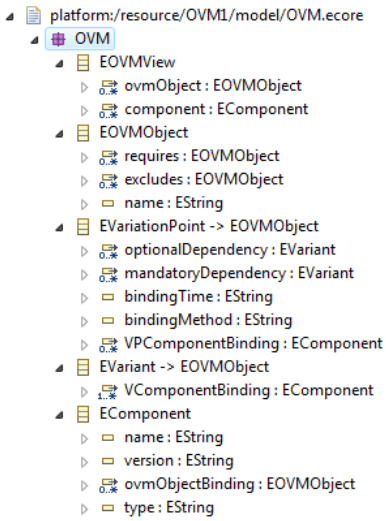


Figure 5.2: Eclipse GMF Dashboard

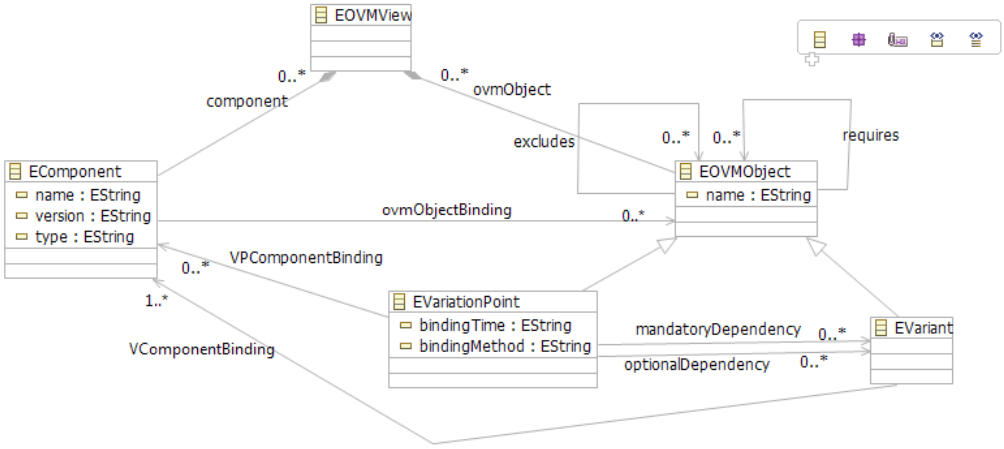
As can be seen from Figure 5.1, the starting point of graphical editor creation using GMF is to define a domain model. The *Create* button in *Domain Model* part of GMF dashboard is used for this purpose. After naming the model and selecting a parent directory, a new blank model is created in the specified directory with the given name. In this model, the domain model elements and relations are defined using either the graphical editor (Figure 5.3(b)) or manually (Figure 5.3(a)). Here, three main elements of domain are defined, namely; variation point, variant and component. As variation point and variant show similar properties, they are shown as generalization relation from *EOVMObject* class. Moreover, the dependencies between these elements are introduced as links in this model.

The second step is to generate the *Domain Gen Model*. The *Derive* link in *Domain Gen Model* box is used for generation. After that, code for classes corresponding to the specified model is generated automatically.

The third step is to generate graphical definition for the model where the rules defined by Pohl [1] is applied. To be more specific, the shapes of variation point, variant and component elements are defined, moreover the dependency relations between elements are configured. A number of built-in shapes like rectangle, ellipse or polygon, are provided and different shapes might be created using polygons. Variant and component shapes are visualized by using the



(a) *Ecore* Model



(b) *Ecore* Diagram

Figure 5.3: Defined *Ecore* Model

built-in shape, rectangle. Direct code modification is also a way to use specialized graphics for elements or links. To create the triangle shape for variation point, a new *Triangle* class is added manually, as it is not supported directly. The link decorations are also selected from built-in styles, such as solid, dashed or dotted line.

The forth step is to develop tooling definition, where a tool palette holding the elements to be used in the graphical model is defined. In the mapping step, the *Ecore*, tooling definition and graphical definition are combined, so the definitions for different parts of the application refer to the same information. After correctly completing the mapping with the help of *Combine* link on dashboard, transformation to *Domain Editor Gen Model* and the code generation for

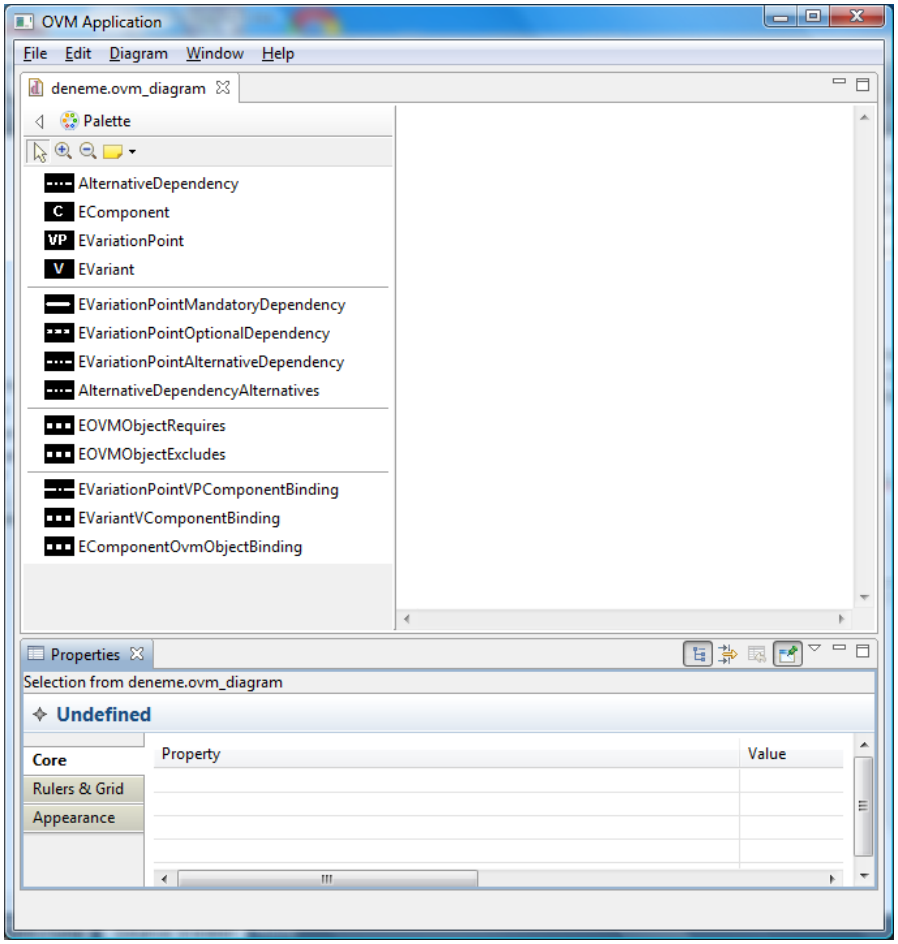


Figure 5.4: OVM Tool

graphical editor are performed and finally the domain specific diagram editor is created.

The tool defined using GMF is shown in Figure 5.4. On the left side, a tool palette is placed. In this palette, the elements that are used to define the OVM are listed together with the component definition and the relations defined in Chapter 4. The *EVariationPoint* and *EVariant* elements are the basics of OVM and *EComponent* is the element representing the components in the component pool. Each element has its own attributes.

As mentioned previously, *EVariant* and *EVariationPoint* elements have some common properties so they both have the same attributes which are

Name: unique name representing the OVM element.

Requires: the list of OVM elements that are required by this element.

Excludes: the list of OVM elements that are excluded by this element.

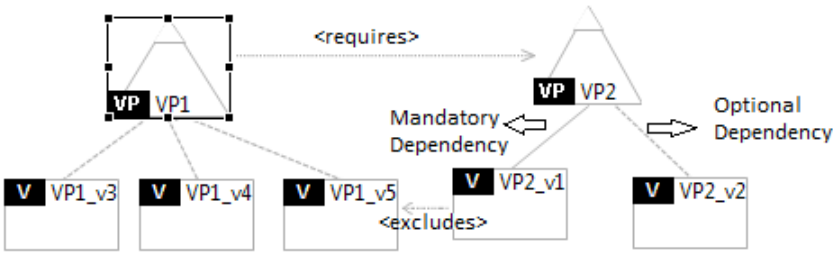


Figure 5.5: Graphical Representation of Elements

Besides the common attributes, both variation point and variant have their own specific attributes. *EVariant* has only one more attribute different than *EVariationPoint* namely,

VComponent Binding: the list of components that realize the variant.

On the other hand, variation point has more attributes specific to itself namely,

Binding Time: the time to bind the variation point.

Binding Method: the method to bind the variability.

Mandatory Dependency: the list of variants those are mandatory.

Optional Dependency: the list of variants those are optional.

Alternative Dependency: the list of variants that are alternatives to each other.

VP Component Binding: the list of components that represent the variation point.

Like OVM elements, *EComponent* element also has attributes defining the component properties. They are listed below:

Name: the name of the component.

Version: the version information for component.

Status: the open/close status of component.

OVM Object Binding: the OVM object list that the component realizes.

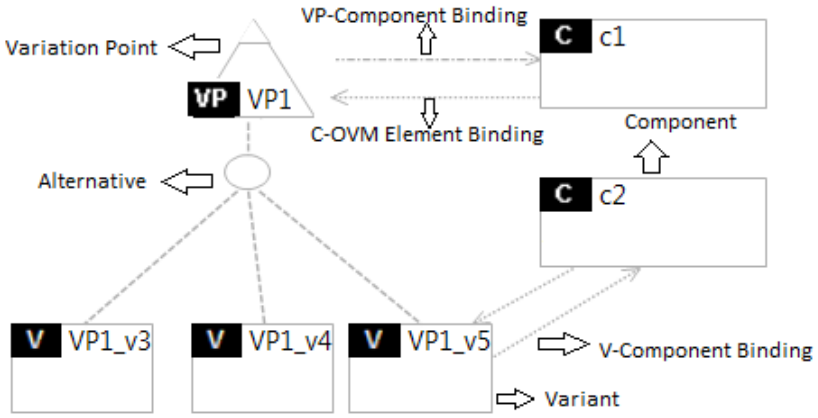


Figure 5.6: Graphical Representation of Elements

The variation point-variant parent-child relationship can be in three different types. Mandatory, optional and alternative are the types of this relationship and they are represented in this OVM tool. The representation of mandatory and optional relations are exactly the same as the one proposed by Pohl [1]. In Figure 5.5, *VP2_v1* variant is mandatory whereas *VP2_v2* variant is optional.

The implementation of alternative dependency is different than the one suggested by Pohl [1]. In order to define an alternative relation, first the alternative dependency from variation point should be created. The relation is created between variation point and *AlternativeDependency* element shown in palette. This element holds the lower and upper bounds of alternative dependency in attributes *min* and *max*, respectively. The alternative variants are then connected to this element using *AlternativeDependencyAlternatives* link in palette. The representation of alternative links is the same as optional dependency as suggested by Pohl. Requires and excludes links are represented similarly, although they mean exactly the opposite as shown in Figure 5.5.

The association links between OVM and components explained in Chapter 4 are implemented by the connections, namely *EVariationPointVPComponentBinding*, *EVariantVComponentBinding* and *EComponentOvmObjectBinding*. The relations from OVM objects to components are defined as artifact dependency in OVM specification [1]. Variation point artifact dependency and its representation is different from variant artifact dependency, therefore they

```
<?xml version="1.0" encoding="UTF-8"?>
<OVM:EOVMView xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns:OVM="OVM">
  <ovmObject xsi:type="OVM:EVariationPoint" name="VP1" VPComponentBinding="//@component.0"
alternativeDependency="//@alternative.0"/>
  <ovmObject xsi:type="OVM:EVariant" name="VP1_v3"/>
  <ovmObject xsi:type="OVM:EVariant" name="VP1_v4"/>
  <ovmObject xsi:type="OVM:EVariant" name="VP1_v5" VComponentBinding="//@component.1"/>
  <ovmObject xsi:type="OVM:EVariationPoint" name="VP2" optionalDependency="//@ovmObject.6"
mandatoryDependency="//@ovmObject.5"/>
  <ovmObject xsi:type="OVM:EVariant" name="VP2_v1"/>
  <ovmObject xsi:type="OVM:EVariant" name="VP2_v2"/>
  <component name="c1" version="1.2" ovmObjectBinding="//@ovmObject.0" type="open"/>
  <component name="c2" ovmObjectBinding="//@ovmObject.3"/>
  <alternative alternatives="//@ovmObject.1 //@ovmObject.2 //@ovmObject.3" min="1" max="3"/>
</OVM:EOVMView>
```

Figure 5.7: XML File Holding Defined Model

are defined as individual links in this tool. The relation in the other way however is new and its representation is same as artifact dependency link of OVM specification [1] (Figure 5.6).

Besides the graphical representation, the GMF also has support to save the OVM in XML format in order to be edited later after creation. The representation in XML file is quite understandable to be parsed by applications different than GMF. This file composes the basis for the following part of the study. Using this OVM definition tool, the goal of creating the final product automatically is achieved.

CHAPTER 6

CASE STUDY IN RADAR GUI DOMAIN

In this chapter, a case study performed to exemplify the proposal made in Chapter 4 is presented. An example domain, namely radar domain, is used in this case study and modeling framework, Eclipse GMF [39], is used as a basis. The organization of this section is as follows: first, the radar domain is explained, and then the use of the Eclipse GMF platform is clarified.

6.1 Radar Domain

The main mission of a RADAR (Radio Detection And Ranging) is to detect various objects, such as: aircraft, ships, spacecraft, guided missiles, vehicles, and even humans. Radio waves are used to detect the objects and basic operation principle is to send radio waves and according to the signals returned from the object, it is detected [42].

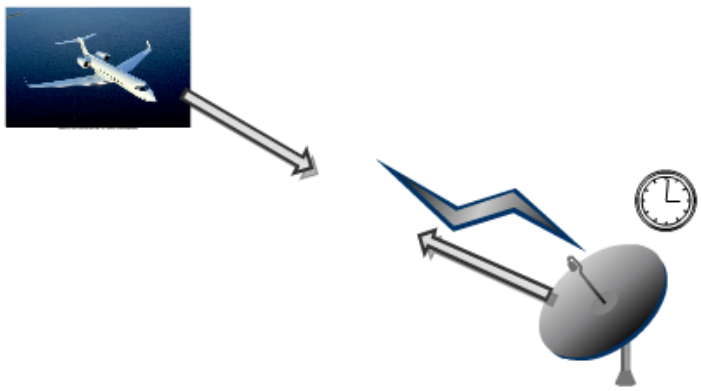


Figure 6.1: Radar Working Principle

Figure 6.2: An Example User Interface for Radar Controls

In different areas, different kinds of radars are used, and one of the basic area for the use of radars is for military purposes. Air defense, ground tracking, surveillance or moving target detection radars are different types of radars used in military. Type of detected objects differs according to the radar's area of usage. Radar is basically composed of a transmitter antenna and a receiver antenna. The radio waves are transmitted by transmitter antenna and if these waves hit a surface, they are reflected back. The reflected waves return to the receiver antenna. By processing the signals returned from the object, the type of the object is identified. The main principle is to measure the time between sending and receiving of the signals and calculating the distance using the speed of light [42].

Radars are used for different purposes, however for all of them there are controlling and monitoring needs. For example, the operator using the radar should be able to monitor the properties of objects that the radar detected (tracks or plots), specify the regions to stop the transmission (blanking sectors) or start and stop the radar operation (start/stop radiation) basically. These properties can be diversified into a very wide range for different purpose radars. They have different use cases, and variability in graphical user interface controls of radars is high, therefore a great variety of graphical user interfaces for radar systems have been designed. Figure 6.2[1] shows an example user interface of an air defense radar. Left side of the screen shows the air picture on which the tracks and blanking sectors are drawn and on the

[1] (c) Copyright ASELSAN 2012. All rights reserved. Used with permission.

right side, the properties of the track are listed.

6.2 OVM for Radar GUI

In Figure 6.3, an example variability model for radar GUI domain is presented. First of all, to clarify the radar domain, explanations for these OVM elements are required.

- *RadarReports* variation point refers to the output supplied by radar which are detection results. Variant *Track* and *Plot* are two types of results, which are the output of different signal processing steps. Plot is the output from an earlier step, and track is the final result.
- *Scope* defines the type of the graphical view for radars, different scopes can be selected based on which type of radar is used.
- *Sector* is used to define and draw the areas that the radar will not radiate to (blanking) or used for tracking of targets (tracking).
- *Map* variation point refers to covered region on earth. Some of the radars might require displaying a map, which can be either raster or vector.
- *SystemState* maps to the functionality of viewing the status of the units consisting of the whole system. This can be presented as a summary or in detail, which are variants *SystemState_Summary* and *SystemState_Detailed* respectively.
- *TimeSettings* variation point is used for time and date settings. *TimeSettings_Manual* is its mandatory variant and it refers manual time/date settings, as the name suggests. *TimeSettings_GPS* is the optional variant and refers setting time/date using a GPS equipment.
- *LocationSettings* variation point represents the functionality of location settings of the radar. It is similar to *TimeSettings* and can be either manual setting or by using GPS.
- *NorthCalibration* variation point stands for the calibration facilities of radar. In order to provide a true detection results for objects, radar should be calibrated to north. There are again two ways to do this, manual which is mandatory, and by using GPS, which is optional.

As can be observed from Figure 6.3, *RadarReports* requires *Scope* variation point, because the radar outputs should be able to be visualized on a view which maps to any kind of *Scope*.

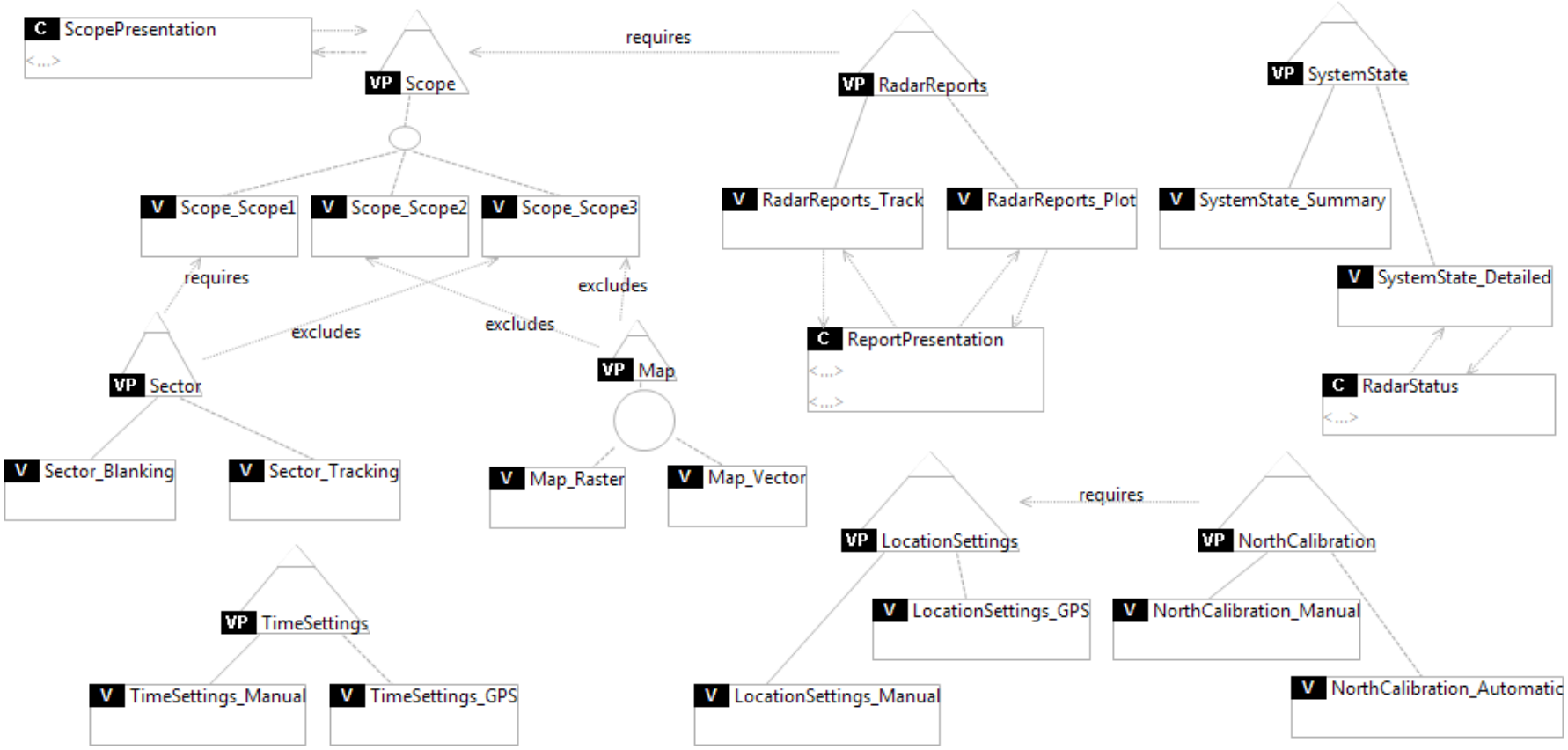


Figure 6.3: Radar GUI OVM

Sector variation point needs also *Scope_Scope1* variant and excludes *Scope_Scope3*, as both types of sectors are not eligible for *Scope3*. *Map* variation point can not be selected with variants *Scope_Scope2* and *Scope_Scope3*, because it is not possible to visualize map regions with these kinds of scope. *NorthCalibration* variation point requires *Location* variation point, in order to correctly determine the north. As the calibration value changes according to the radar location, without location information, it is not possible to perform calibration.

In this model, three components are shown, namely, *ScopePresentation* representing *Scope* variation point, *ReportPresentation* realizing both *RadarReports_Track* and *RadarReports_Plot* variants and *RadarStatus* implementing the variant *SystemStatus_Detailed*. It might be wondered why every OVM element in this model do not map to at least one component; the answer is to keep this example simple and understandable. In the following section, adding a brand new component realizing a variant and adding a new version of a component are exemplified.

6.3 Component Realization

Throughout the case study, reusable components are realized in Open Services Gateway Initiative [43] environment as OSGi bundles. Therefore, Java and OSGi terminologies are used. OSGi is a framework that supports dynamic component model and provides an infrastructure for adding or removing components dynamically at runtime. Bundle is the term used for deployment in OSGi instead of the more generic term, component. A bundle specifies its requirements in a manifest file by using specific parameters. A manifest file basically includes name, version and dependency information for the component and is located in *META-INF* directory with the name *MANIFEST-MF* [44].

There are more parameters that can be added optionally in a manifest file, but only the ones which are interested are explained below and an example manifest file is shown in Figure 6.4.

Export-Package specifies the packages that this bundle exports. Only the packages specified here can be available to other bundles.

Import-Package specifies the packages imported from other bundles. The packages specified here are visible in this bundle.

Bundle-SymbolicName is a must and holds a unique identifier for bundle.

Bundle-Version is the version information for bundle.

Figure 6.4: Example Manifest File

Bundle-Name is similar to Bundle-SymbolicName, but a more convenient one.

Bundle-Activator specifies the class that is responsible from both starting and stopping this bundle.

Require-Bundle specifies the bundles which are required in the runtime environment of this component.

OSGi platform has the concept of target platform. Products composed of OSGi bundles has a target platform which is basically a directory that holds jar files of bundles used in the product.

6.4 Usage Scenarios

Starting point is creating the OVM of radar GUI domain. This step belongs to domain engineering process when the domain is analyzed. Both common and variable parts of the domain are expressed during this process and configurable parts are modeled by using OVM technique. OVM tool defined in Chapter 5 supports the definition of this variability model. After creating the model using this tool, next step is the realization process, which is again a part of domain engineering step.

The components are developed as OSGi bundles. After that the component information should be inserted into the model defined using this tool when the variability model is created. As mentioned before, component definition is a functionality supported by this tool. To add this information, a new component element is created having the same name and version with the developed OSGi bundle. The OVM element bindings are also created by adding OVM el-

ement binding relation to this component. The same process is repeated when a new version of a component is created.

Another aspect is automatic product derivation from this association model. Before starting product derivation, the OVM elements' component mapping information should be gathered. In the previous step, components' OVM element binding relations have been created; by using those links the required component mapping information is derived. Besides deriving the links of OVM-to-components, they can also be added manually like creating the component-to-OVM links. Next, the relations from OVM to components are used to find the components that will be included in the final product. There are also implicit dependencies embedded in component's manifest file. The *Require-Bundle* header of manifest file is critical in finding the required components (bundles) that should be included in the final product. By looking at the Require-Bundle property of the components' manifest files, required components are selected automatically if they are not already added as a result of OVM element selection.

The components, specifically, OSGi bundles are kept in a directory as jar files and named using the component's name and version. In the previous step, component names and versions have been collected which are now searched in the component directory. At this point, it is assumed that component names in this directory and in the association model are consistent. After the selected components are found in this directory, they are copied to the target platform of final product. As expressed before, core components that should be included in all products of that domain are assumed to be included too.

When the product derivation is completed, analysis operations can also be conducted reporting the following information: 1. the bundles and their versions, 2. all variation points, 3. the variants, 4. the unbound variation points, 5. the bound variation points, 6. bundles that can not be found, 7. bundles that are selected as a result of components' internal dependencies.

6.5 Product Derivation Tool

The OVM tool which supports variability model definition has been explained in Chapter 5. In the scope of this study, product derivation process is also supported by a tool which is strongly connected to the OVM tool, because the model defined using the OVM definition part is used in product derivation support tool.

This tool is specific to OSGi projects. OSGi builds need a configuration file (*config.ini*) and component jar files in deployment environment. Therefore, this tool aims at producing *config.ini* file. The process defined in Section 4.5 is implemented and the starting point is selecting the OVM file produced by the OVM tool and final output is the *config.ini* file belonging to the product. This file holds the configuration that is specific to the product. The OSGi bundles composing the product and their starting sequences are defined in this file. As OSGi bundles are services, during the application launch, the services are started sequentially. Moreover, OSGi parameters are also defined in this file.

The process is defined in the following steps and illustrated in Figure 6.5

1. The OVM file which is generated by OVM tool, is selected by using the GUI provided. The variation points and variants are listed as a tree whose nodes are selectable.
2. The component directory is selected. OSGi bundles of the domain, which is referred as component pool or repository previously, are saved in this directory. The configuration management integration is not supported yet, but components can be gathered from CM tool in the future.
3. The next step is finding the component corresponding to the selected variations. At this point, the requirements of the selected variations are also considered. Components found in the specified directory are listed and the ones corresponding to variations are shown as selected. Manual selection is also possible, because the common components should be selected manually which stands as the reference architecture for the product. The start sequences should also be specified.
4. Target platform directory is selected, which means the deployment directory. The product's *config.ini* is created under this directory.
5. Final step is the *config.ini* file creation and copying the components to the target platform specified.

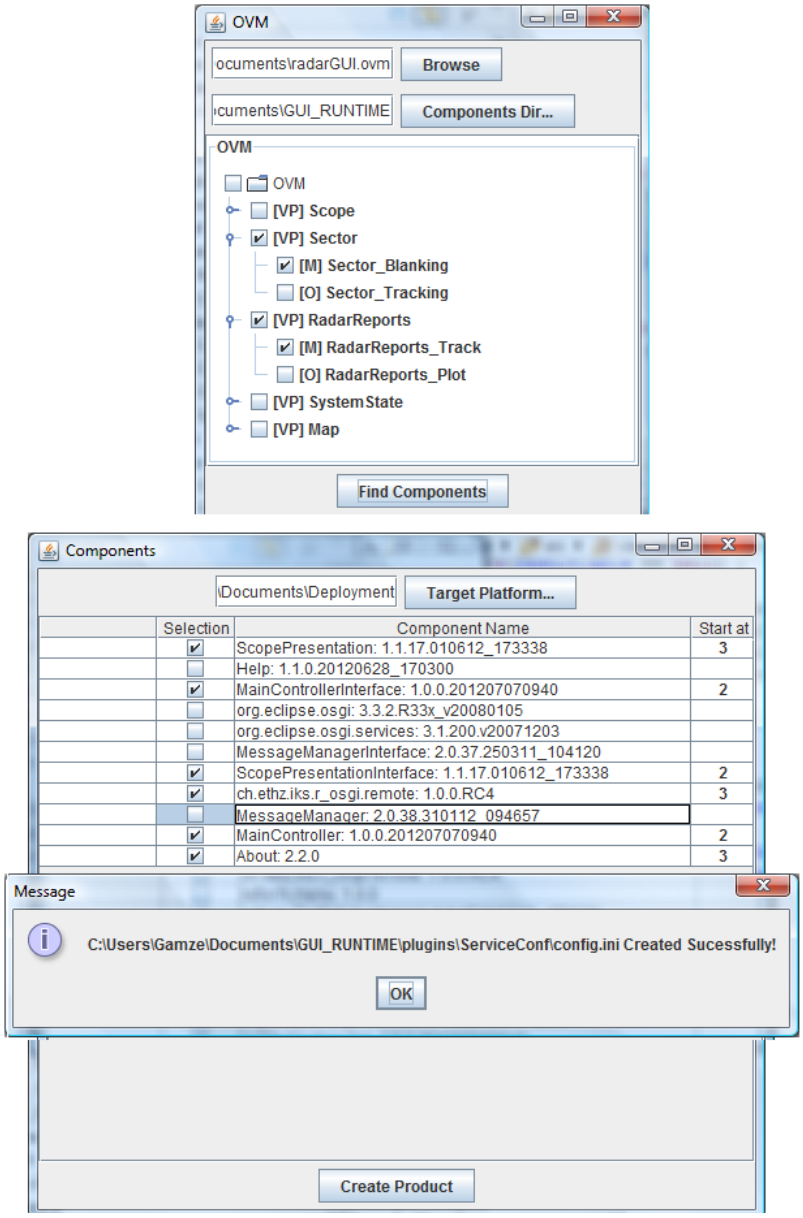


Figure 6.5: Product Derivation Tool

CHAPTER 7

CONCLUSION AND DISCUSSION

In this chapter, first the discussions are proposed and then the results of this study are summarized.

7.1 Discussion

There are some issues to be considered about this study, which might be used to improve it according to the items mentioned below.

- The components defined in this tool should be kept consistent with the actual components. It is hard to ensure consistency manually, so providing component information to the OVM-component association model might be automated. The component information can be obtained from the CM tool and transferred to this model.
- The consistency in the other way is also critical. When creating a new component, variability information, specifically variation points and variants that are realized should be inserted into the CM tool. Collecting this information from this OVM tool should also be automated, as manual definition is prone to errors.
- In this tool, binding time and method attributes are defined specifically to variation points with the assumption that these attributes are specific to variation points, not specialized according to components. If these are component specific, this implementation should be modified.
- In order to implement the analysis operations by using the relations among OVM elements and components efficiently, they can be represented in a relational database. By doing so, the complexity of deriving the variation information of a component or the

components realizing a variation might be lowered.

7.2 Conclusion

In this thesis, the method of representing variability information in configuration items is proposed. Variability information is critical to manage different configurations of products. By adding this information to configuration items, traceability of variability among versions is achieved. Another aspect of relating variability and component is to automate product derivation process. In order to provide tool support for the method proposed here, OVM tool is developed, and used in a case study of radar GUI domain.

From OVM perspective, binding time, method and component list attributes are introduced. It is explained that binding time is critical information for variability, therefore explicit representation of it brings in the advantage of automatic analysis and also decreases the errors. Binding method attribute can also be used as supporting information. On the other hand, component list attribute provides first-class representation for realization, which facilitates the automatic product derivation process.

From configuration items, namely components perspective, status and OVM element list attributes are defined. Status stands for the direct summary of the binding information. OVM element list holds the variability realized in that component, which is main focus of the study. This information can be used both for variability analysis and also for tracing the variability between subsequent versions.

As a future work, the analysis operations on OVM can be stated. As the OVM includes all constraint dependencies, any invalid configuration of selected variations can be detected. It is also possible to analyze the possible combinations of variations and to list available product configurations.

REFERENCES

[1] K. Pohl, and L. Northrop, *Software Product Line Engineering Foundations, Principles, and Techniques*, Springer, 2005.

[2] P. Clements, and L. Northrop, *Software Product Lines: Practices and Patterns*, US: Addison-Wesley, 2002.

[3] C. Krueger, "Variation Management for Software Production Lines", *Proc. of the 2nd International Software Product Line Conference*, 2002.

[4] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering", *IEEE Software*, vol.19, no. 4, 2002, pp. 58-65.

[5] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A Framework for Modelling Variability in Software Product Families", *Proceedings of the Third Software Product Line Conference* (SPLC'04), 2004.

[6] S. Bühne, K. Lauenroth, and K. Pohl, "Why is it not Sufficient to Model Requirements Variability with Feature Models?", *Proceedings of the Workshop: Automotive Requirements Engineering* (AURE'04), co-located at RE'04, Nagoya, Japan, 2004.

[7] R. van Ommering, "Configuration Management in Component Based Product Populations", *Proceedings of the 2001 ICSE Workshops on SCM*, 2001, pp. 16-23.

[8] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, March 2000, pp. 78-85.

[9] C. Brink, M. Peters, and S. Sachweh, "Improved Software Project Management Through The Use of Software Product Lines", *Proc. of First International Scientific Conference on Project Management in the Baltic Countries*, Riga, University of Latvia, 2012.

[10] F. Heidenreich, J. Kopcsek, and C. Wende, "FeatureMapper: Mapping Features to Models", *Companion Proceedings of the 30th International Conference on Software Engineering* (ICSE'08), New York, NY, USA, ACM, 2008.

[11] F. Heidenreich, P. Sánchez, J. Santos, S. Zschaler, M. Alférez, J. Araújo, L. Fuentes, U. Kulesza, A. Moreira, and A. Rashid, "Relating Feature Models to Other Models of a Software Product Line A Comparative Study of FeatureMapper and VML", *Transactions on Aspect-Oriented Software Development VII: a Common Case Study for Aspect-Oriented Modeling*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 69-114.

[12] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J. C. Royer, A. Rummler, and A. Sousa, "A Model-Driven Traceability Framework for Software Product Lines", *The Journal of Software and Systems Modeling*, vol. 9, no. 4, 2010, pp. 427-451.

[13] F. Roos-Frantz, D. Benavides, A. Ruiz-Corte's, A. Heuer, and K. Lauenroth, "Quality-Aware Analysis in Product Line Engineering with the Orthogonal Variability Model",

Software Quality Journal Special Issue on Quality Engineering for Software Product Lines, vol. 20, no. 3-4, 2012, pp. 519-565.

[14] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, Reading, Massachusetts, 2000.

[15] F. Van der Linden, "Software Product Families in Europe: The Esaps & Caf'e Projects", *IEEE Software*, vol. 19, no. 4, 2002, pp. 41-49.

[16] D. M. Weiss, and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] Object Management Group (OMG), *Reusable Asset Specification*, Final Adopted Specification, PTC/04/06/06, June 2004.

[18] L. Yu, and S. Ramaswamy, "A Configuration Management Model for Software Product Line", *INFOCOMP Journal of Computer Science*, 2006, pp. 82-88.

[19] IEEE Computer Society, *IEEE Standard for Configuration Management in Systems and Software Engineering*, IEEE Std 828TM-2012 (Revision of IEEE Std 828-2005), 2012.

[20] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat, "Variability in Time - Product Line Variability and Evolution Revisited", *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems* (VAMOS 2010), 2010.

[21] D. Benavides, S. Segura, and A. Ruiz-Cort'es, "Automated Analyses of Feature Models 20 Years Later: a Literature Review", *Information Systems*, vol. 35, no. 6, 2010, pp. 615-636.

[22] E. D. de Souza Filho, R. de Oliveira Cavalcanti, D. F. S. Neiva, T. H. B. de Oliveira, L. B. Lisboa, E. S. de Almeida, and S. R. de Lemos Meira, "Evaluating Domain Design Approaches Using Systematic Review", *Proceedings of Second European Conference on Software Architecture* (ECSA 2008), Paphos, Cyprus, vol. 5292. Springer, 2008, pp. 50-65.

[23] W. B. Frakes, and K. C. Kang, "Software Reuse Research: Status and Future", *IEEE Transactions on Software Engineering*, vol. 31, no. 7, 2005, pp. 529-536.

[24] L. Chen, M. A. Babar, and N. Ali, "Variability Management in Software Product Lines: a Systematic Review", *Proceedings of the 13th International Software Product Line Conference* (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, 2009, pp. 81-90.

[25] K. Lee, K. C. Kang, and J. Lee, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering", *Proceedings of the 7th International Conference on Software Reuse* (ICSR-7), London, UK: Springer-Verlag, 2002, pp. 62-77.

[26] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Ann. Softw. Eng.*, vol. 5, 1998, pp. 143-168.

[27] M. L. Griss, J. Favaro, and M. d' Alessandro, "Integrating Feature Modeling with the RSEB", *Proceedings of the 5th International Conference on Software Reuse* (ICSR-98), IEEE Computer Society, 1998, pp. 76-85.

[28] M. Eriksson, J. Börstler, and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations", *Proceedings of 9th International Software Product Lines Conference* (SPLC 2005), Rennes, France, September 26-29, 2005, pp. 33-44.

[29] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley-Longman, May 1997.

[30] S. Bühne, G. Halmans, and K. Pohl, "Modeling Dependencies between Variation Points in Use Case Diagrams", *Proceedings of the 9th International Workshop on Requirements Engineering-Foundation for Software Quality* (REFSQ'03), Klagenfurt/Velden, Österreich, June, 2003.

[31] T. Von der Massen, and H. Lichter, "Modeling Variability by UML Use Case Diagrams", *Proceedings of the International Workshop on Requirements Engineering for Product Lines* (REPL'02), 2002.

[32] G. Halmans, and K. Pohl, "Communicating the Variability of a Software Product Family to Customers", *Software and Systems Modeling*, vol. 2, no. 1, March 2003.

[33] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl, "Variability Issues in Software Production Lines", *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, Heidelberg, Germany: Springer-Verlag, 2001.

[34] M. Svahnberg, J. v. Gurp, and J. Bosch, "On the Notion of Variability in Software Product Lines", *Proceedings of the Working IEEE/IFIP Conference on Software Architecture* (WICSA 2001), Amsterdam, The Netherlands, 2001, pp. 45-55.

[35] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato,*Version Control with Subversion*, Sebastopol, CA: O'Reilly, 2004.

[36] T. Morse, *CVS*, Linux J., Issue 21, Article 3, 1996.

[37] J. Loeliger, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*, O'Reilly, 2009.

[38] IBM Corp.: IBM Rational ClearCase, http://www.ibm.com/software/awdtools/clearcase/, Last visited: 28.08.2012.

[39] Graphical Modeling Framework - Eclipsepedia, http://wiki.eclipse.org/GMF, Last visited: 28.08.2012.

[40] Steinberg D., Budinsky F., Paternostro M., Merks E., *EMF: Eclipse Modeling Framework*, 2nd Edition, Addison-Wesley Professional, 2008.

[41] Graphical Editing Framework - Eclipsepedia, http://wiki.eclipse.org/GEF, Last visited: 28.08.2012.

[42] M. I. Skolnik, *Introduction to Radar Systems*, McGraw-Hill, New York, 2002.

[43] OSGi Alliance Technology The OSGi Architecture, http://www.osgi.org/Technology/WhatIsOSGi, Last visited: 28.08.2012.

[44] Understanding The Manifest, http://java.sun.com/developer/Books/javaprogramming/JAR/basics/manifest.html, Last visited: 28.08.2012.

APPENDIX A

USER MANUAL

A.1 OVM Tool Elements

OVM Tool provides a graphical user interface to define OVM and components with the dependencies between them. There are three main parts of the tool; tool palette, diagram and properties window. In the tool palette, the elements used to define the model are listed. In the diagram part, graphical model is drawn and in properties window, the element's properties are listed and editing can be performed.

Variation point and variant are two basic elements of OVM as explained in Chapter 3 and they are referred as *EVariationPoint* and *EVariant* respectively. Their graphical representations and attributes are shown in Figures A.2 and A.3. The component element that can be defined is named as *EComponent* as shown in Figure A.4.

The variability dependencies, mandatory, optional and alternative are shown in Figures A.5 and A.6 and introduced by using *EVariationPointMandatoryDependency* and *EVariationPointOptionalDependency* elements in palette. In order to add alternative dependency, first an alternative dependency element added using *AlternativeDependency* as a new element and connected to parent variation point with *EVariationPointAlternativeDependency*. Alternative variants are then added to this *AlternativeDependency* element with the connection *AlternativeDependencyAlternatives*. The cardinality of alternative dependency is defined in the attributes of *AlternativeDependency* element which are min and max attributes.

The requires and excludes constraint dependencies are shown as *EOVMObjectRequires* and *EOVMObjectExcludes*. The OVM element-Component relations are shown in Figure A.8. *EVariationPointVPComponentBinding*, *EVariantVComponentBinding* and *EComponentOV-*

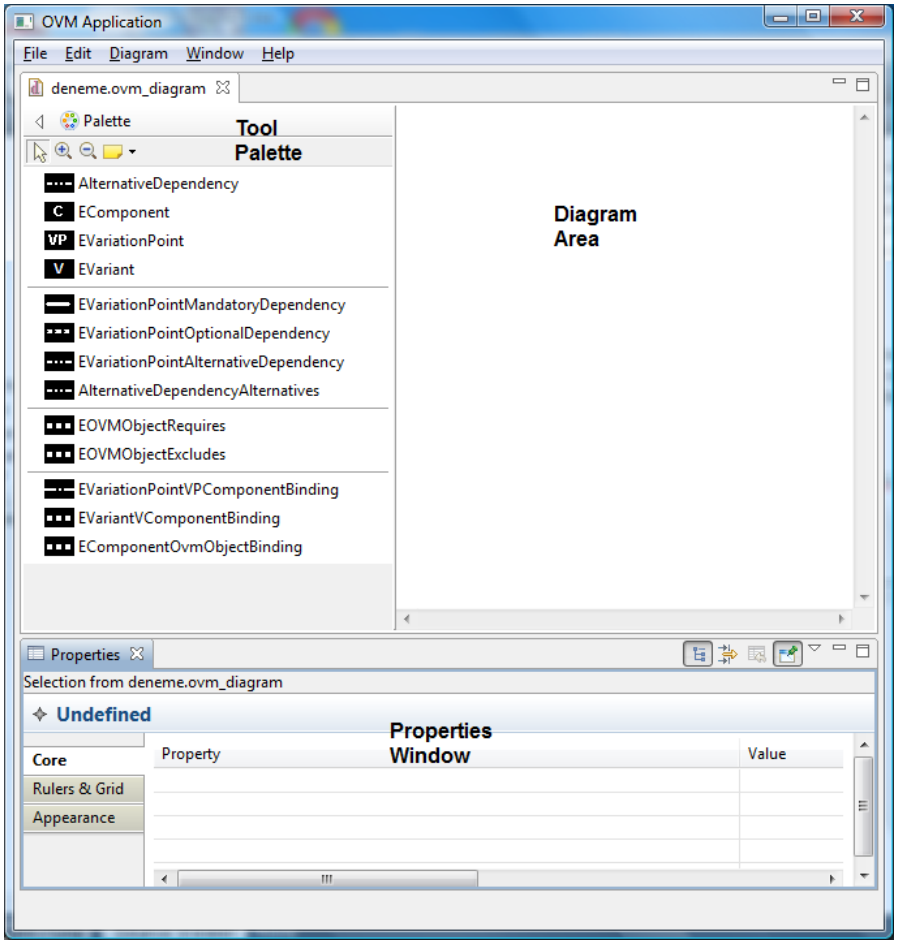


Figure A.1: OVM Tool

MObjectBinding define these relations.

A.2 Operations

OVM tool provides a number of operations that are essential, which are element definition, editing, deletion or observing the element information.

Elements can be added using 3 different ways.

1. Selecting the element from tool palette.
2. Clicking on the diagram and selecting from the pop-up menu appeared.
3. Selecting from the menu opened when adding a relation.

Editing operations on the model can be done either on the diagram or from the properties

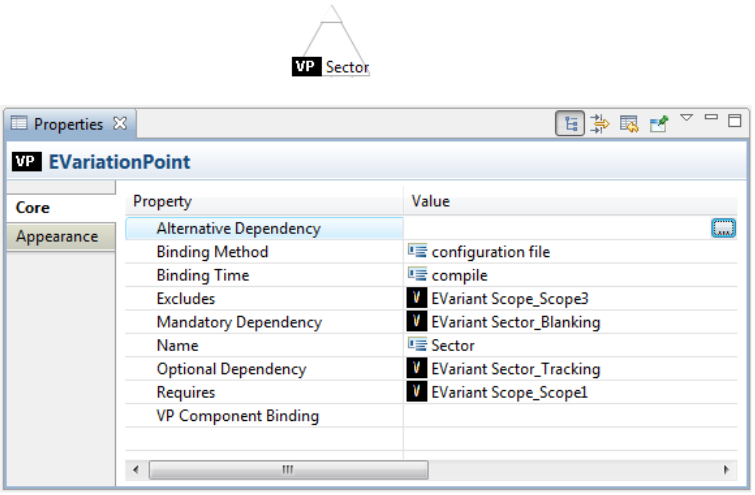


Figure A.2: EVariationPoint Graphical Representation and Attributes

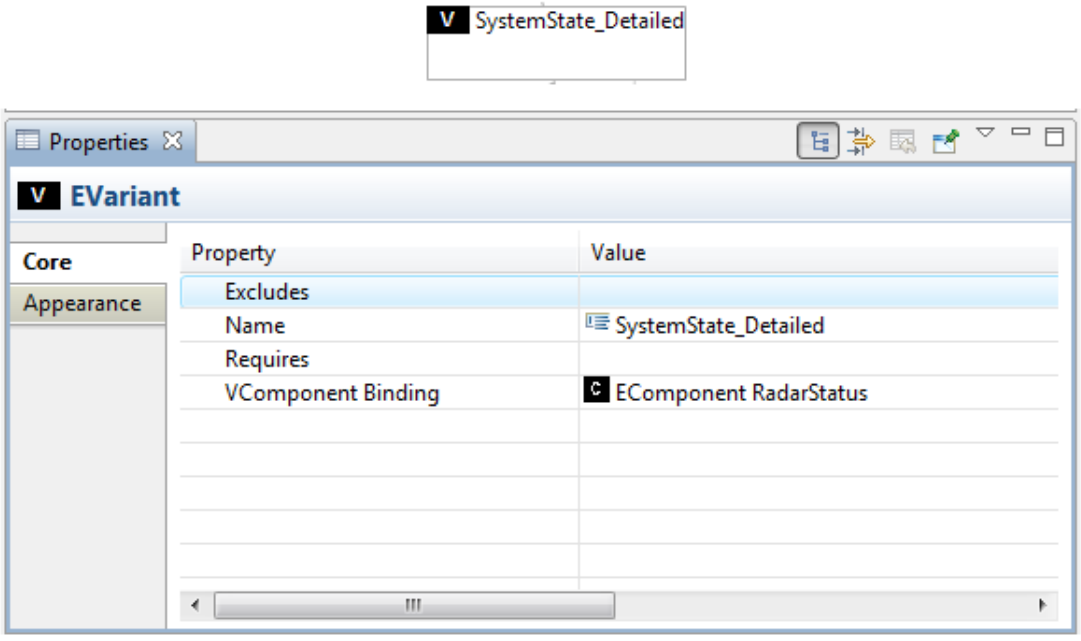


Figure A.3: EVariant Graphical Representation and Attributes

window. All the attributes of OVM elements and components can be edited from properties window. Also, when a new relation is added to an element in the diagram, the attributes holding the relations are automatically updated. Besides editing, the attributes can also be observed from the same window.

Any relation provided in this tool, can be created by selecting from the tool palette. After

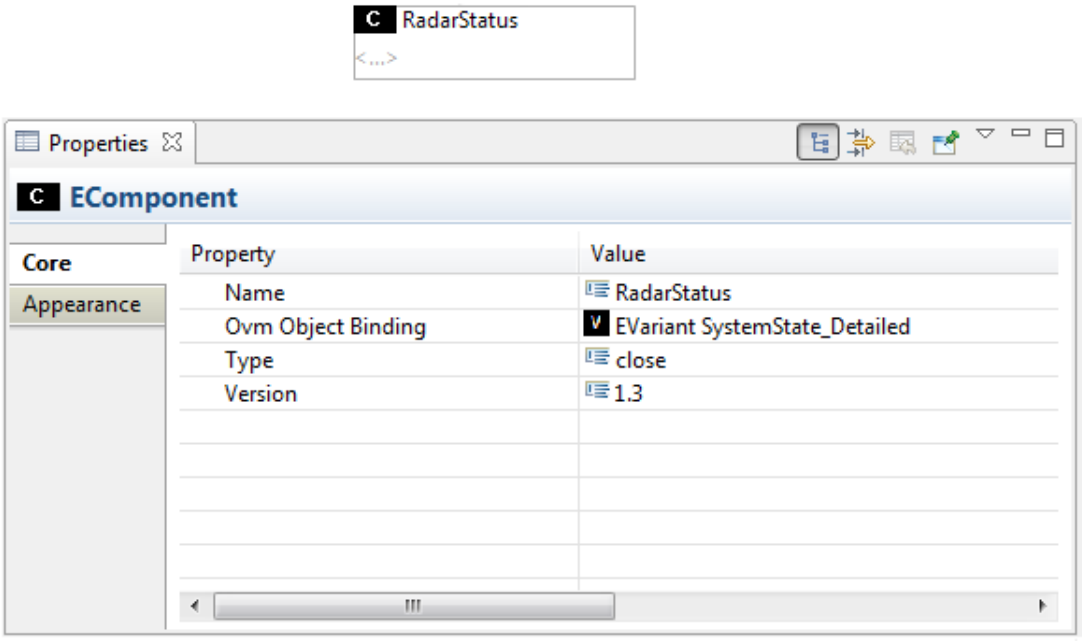


Figure A.4: EComponent Graphical Representation and Attributes

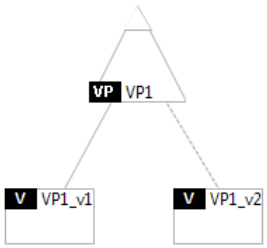


Figure A.5: Mandatory and Optional Relations

the desired relation is selected, the parent element is clicked and dragged till the child and released. The relation information is then added automatically to corresponding attribute.

All delete operations are performed by selecting from the menu appearing with a right mouse click or *Del* key (Figure A.10).

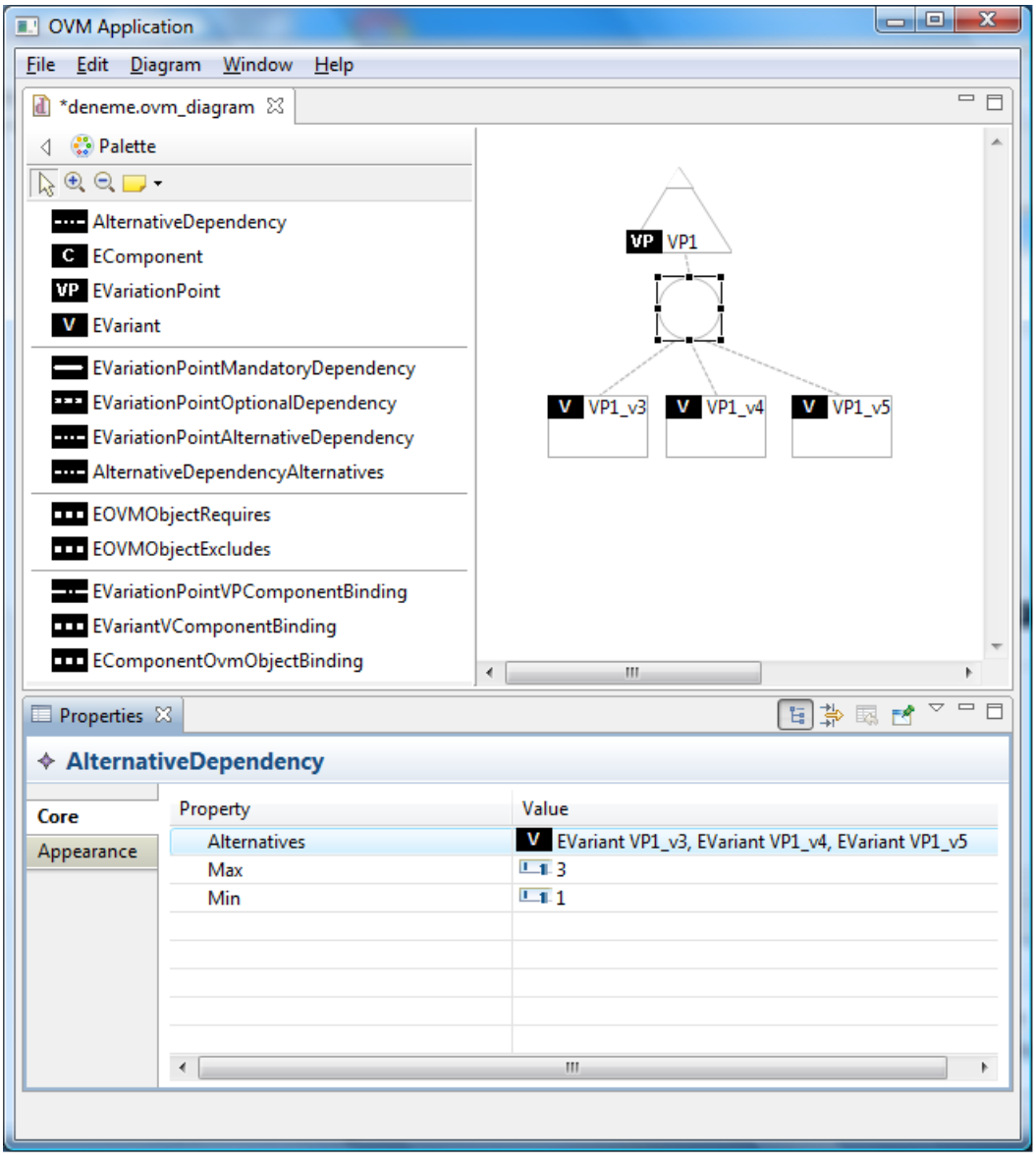


Figure A.6: Alternative Relation

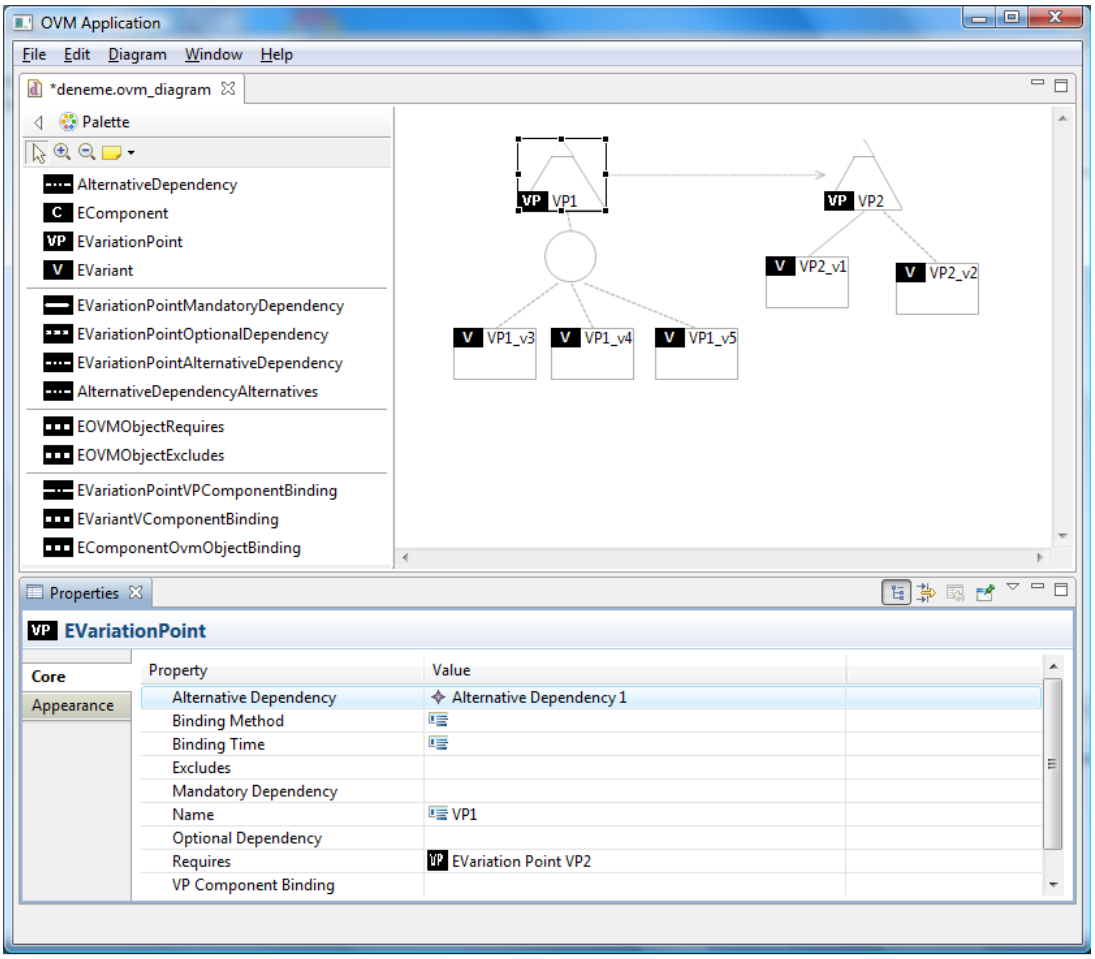


Figure A.7: Requires Link

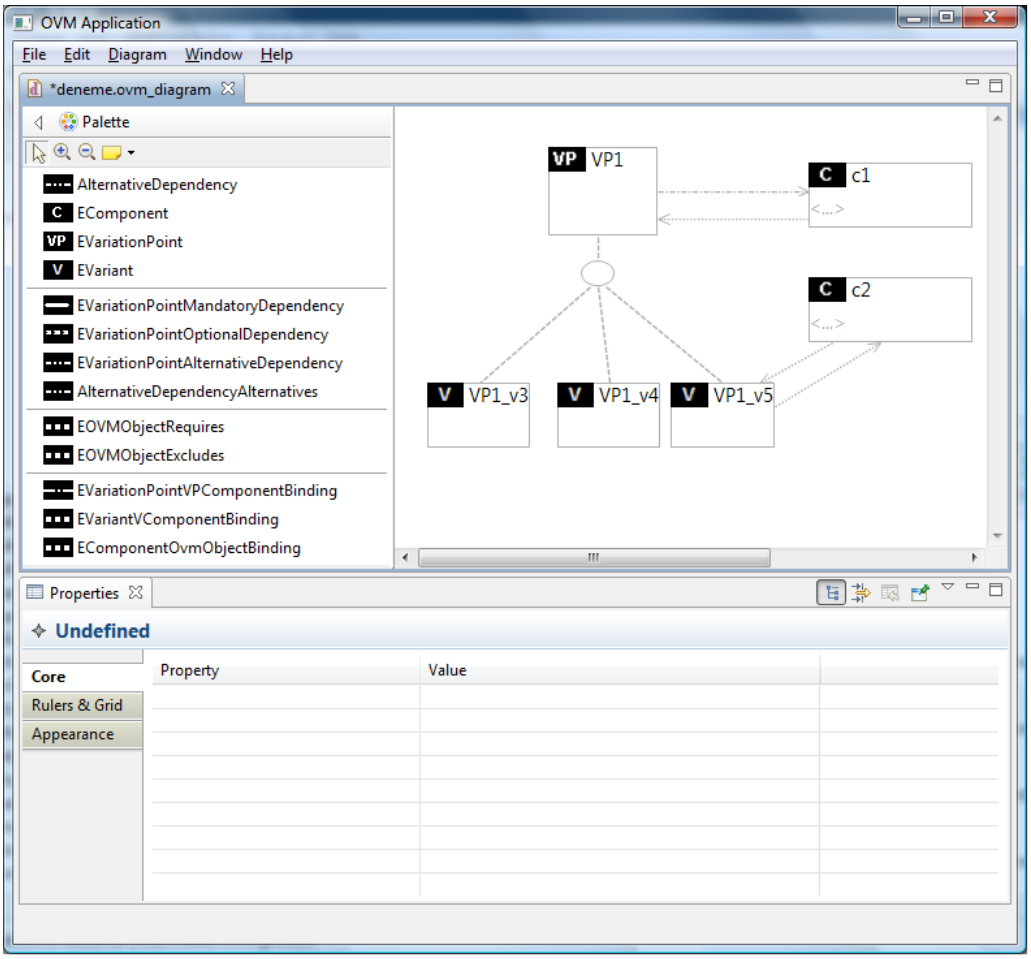


Figure A.8: OVM Objects-Component Links

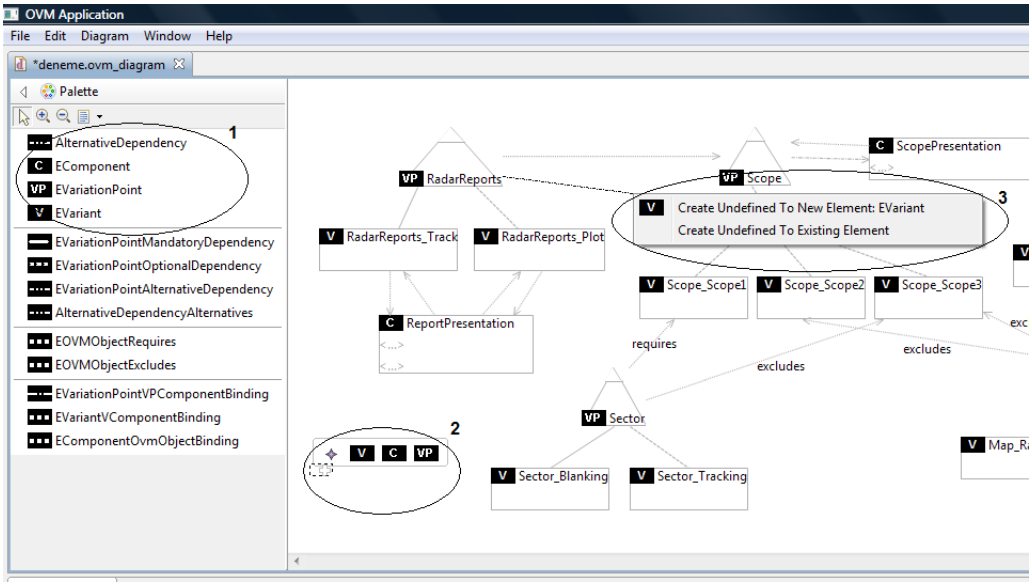


Figure A.9: Three Ways of Creating a New Element

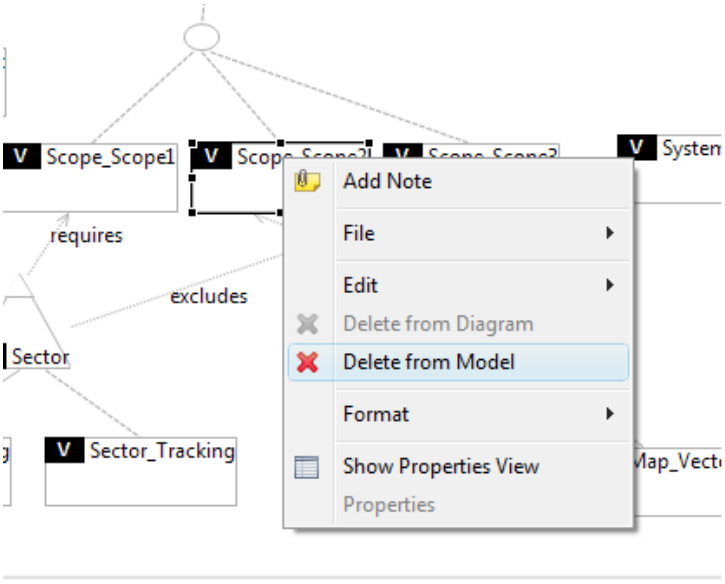


Figure A.10: Deletion of an Element from Model