ARCHITECTURE CONFORMANCE ANALYSIS IN SOFTWARE PRODUCT LINE ENGINEERING USING REFLEXION MODELING

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Onur Özdemir

December 2015

ARCHITECTURE CONFORMANCE ANALYSIS IN SOFTWARE PRODUCT LINE ENGINEERING USING REFLEXION MODELING

By Onur Özdemir

December 2015

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Bedir Tekinerdoğan(Advisor)

Ali Hikmet Doğru

Can Alkan

Approved for the Graduate School of Engineering and Science:

Levent Onural
Director of the Graduate School

ABSTRACT

ARCHITECTURE CONFORMANCE ANALYSIS IN SOFTWARE PRODUCT LINE ENGINEERING USING REFLEXION MODELING

Onur Özdemir
M.S. in Computer Engineering
Advisor: Bedir Tekinerdoğan
December 2015

Software product line engineering (SPLE) aims to provide pro-active, pre-planned reuse at a large granularity (domain and product level) to develop applications from a core asset base. By investing upfront in preparing the reusable assets, it is expected to develop products with lower cost, get them to the market faster and produce with higher quality. In alignment with these goals different SPLE processes have been proposed that usually define the SPLE process using the two lifecycles of domain engineering and application engineering. In domain engineering a reusable platform and product line architecture is developed. In application engineering the results of the domain engineering process are used to develop the product members.

One of the most important core assets in SPLE is the software architecture. Hereby we can distinguish between the product line architecture and application architecture. The product line architecture is developed in the domain engineering process and represents the reference architecture for the family of products. The application architecture represents the architecture for a single product and is developed by reusing the product line architecture. It is important that the application architectures remain consistent with the product line architecture to ensure global consistency. However, due to evolution of the product line architecture and/or the application architecture inconsistencies might arise leading to an architecture drift. In the literature several architecture conformance analysis approaches have been proposed but these have primarily focused on checking the inconsistencies between the architecture and code. Architecture conformance analysis within the scope of SPLE has not got much attention.

In this thesis we first present the results of our tertiary systematic literature

review to systematic reviews on software product line testing. Subsequently, we propose a systematic architecture conformance analysis approach for detecting inconsistencies between product line architecture and application architecture. For supporting the approach we adopt the notion of reflexion modeling in which architecture views of product line architecture are compared to the architecture views of the application architecture. For illustrating our approach we use the *Views and Beyond* approach together with a running case study. Furthermore, we present the provided tool support for the presented approach. Our evaluation shows that the approach and the corresponding tool are effective in identifying the inconsistencies between product line architectures and application architectures.

Keywords: Systematic Literature Review, Tertiary Study, Software Architecture Viewpoints, Architecture Conformance Analysis, Software Product Line, Reflexion Modeling.

ÖZET

YAZILIM ÜRÜN HATTI MÜHENDİSLİĞİNDE YANSIMA MODELLEMESİ KULLANILARAK MİMARİ UYUM ANALİZİ

Onur Özdemir
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Bedir Tekinerdoğan
Aralık 2015

Yazılım ürün hattı mühendisliği (YÜHM), çekirdek varlık temelinden uygulama geliştirmek için geniş çapta (ortak alan ve ürün seviyesinde) ileriye yönelik, planlı yeniden kullanım sağlanmasını hedefler. Yeniden kullanılabilir varlıkların hazırlanmasına yapılan ön yatırım ile ürünlerin daha düşük maliyet ile geliştirilmesi, onların pazara daha hızlı sunulması ve üretimlerinde kalitenin artması beklenir. Bu hedefler doğrultusunda farklı YÜHM süreçleri önerilmiştir. Önerilen bu süreçler, genel olarak YÜHM sürecini, temel mühendisliği ve uygulama mühendisliği şeklindeki iki üretim döngüsünü kullanacak şekilde tanımlar. Temel mühendisliğinde yeniden kullanılabilir bir platform ve ürün hattı mimarisi geliştirilir. Uygulama mühendisliğinde ürün üyelerinin geliştirilmesi için temel mühendisliği sürecinin sonuçları kullanılır.

Yazılım mimarisi, YÜHM'ndeki en önemli çekirdek varlıklarından biridir. Bundan dolayı ürün hattı mimarisi ile uygulama mimarisi ayrımını yapabiliriz. Ürün hattı mimarisi, temel mühendisliği sürecinde geliştirilir ve aynı familyaya ait ürünler için referans mimarisini temsil eder. Uygulama mimarisi, tek bir ürün için mimariyi temsil eder ve ürün hattı mimarisinin yeniden kullanılması ile geliştirilir. Genel tutarlılığın sağlandığından emin olmak için, uygulama mimarilerinin ürün hattı mimarisi ile uyumlu kalması önemlidir. Ancak ürün hattı mimarisinin ve/veya uygulama mimarisinin gelişiminden dolayı, bir mimari sapmaya neden olacak uyumsuzluklar ortaya çıkabilir. Literatürde birkaç mimari uyum analiz yöntemi önerilmiş olsa da, bunlar öncelikli olarak mimari ve kod arasındaki uyumsuzlukların denetlenmesine odaklanmıştır. YÜHM kapsamında, mimari uyum analizi pek fazla ilgi görmemiştir.

Bu tezde, ilk olarak, yazılım ürün hattı test sürecini inceleyen sistematik

taramaların değerlendirildiği üçüncül sistematik literatür taramamızı sunuyoruz. Daha sonra, ürün hattı mimarisi ile uygulama mimarisi arasındaki uyumsuzlukları ortaya çıkaracak sistematik bir mimari uyum analiz yöntemi öneriyoruz. Yöntemi desteklemek için, kapsamında, ürün hattı mimarisinin mimari görünümleri ile uygulama mimarisinin mimari görünümlerinin karşılaştırıldığı, yansıma modellemesi kavramını benimsiyoruz. Yöntemimizi daha iyi açıklamak için, *Görümler ve Ötesi* yöntemini çalışan bir örnek-olay incelemesi ile birlikte kullanıyoruz. Üstelik, verilen yöntem için sağlanan araç desteğini de sunuyoruz. Değerlendirmemiz, yöntem ve ilgili aracın, ürün hattı mimarileri ile uygulama mimarileri arasındaki uyumsuzlukların belirlenmesinde etkili olduklarını gösteriyor.

Anahtar sözcükler: Sistematik Literatür İnceleme, Üçüncül İnceleme, Yazılım Mimari Bakış Açıları, Mimari Uyum Analizi, Yazılım Ürün Hattı, Yansıma Modellemesi.

Acknowledgement

I would like to express my sincerest gratitude to my advisor Bedir Tekinerdoğan for his support, time and guidance throughout my thesis. Without his constructive advices, this thesis would not have been completed.

I am also thankful to Can Alkan and Ali Hikmet Doğru for kindly accepting to be in the thesis committee and also for giving their valuable time to read and review this thesis.

Finally, I dedicate this thesis to Mustafa Kemal Atatürk, Father of the Turks, who is a liberator, great hero and symbol of a nation. Thanks to his unsurpassed principles and reforms upon which modern Turkey was established, today we have high level of scientific education in our universities. It goes without saying that "our true mentor in life is science", as he underlines.

With all my eternal gratitude, respect and longing to Atatürk who will live on in our hearts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software product line engineering (SPLE) aims to provide pre-planned reuse at a coarse-grained level, such as domain and product level, in order to develop software products from a core asset base. By investing upfront in preparing the reusable assets, it is expected to advance productivity through automation, improve software quality and achieve high level of reuse in all development phases [2]. In alignment with these goals different SPLE processes have been proposed that usually define the SPLE process using the two lifecycles of domain engineering and application engineering. In domain engineering a reusable platform and product line architecture (also can be named as reference architecture, common architecture or domain architecture) is developed [3]. In application engineering the results of the domain engineering process are used to develop the product members [3].

1.1 Problem Statement

One of the most important core assets in SPLE is the software architecture. By means of this we can distinguish between the product line architecture and application architecture. The product line architecture is developed in the domain

engineering process and represents the reference architecture for the family of products. The application architecture represents the architecture for a single product and is developed by reusing the product line architecture in the application engineering process.

The product line architecture is mainly responsible to provide commonality to software artifacts that will later be integrated with the variability introduced in the product line to develop a concrete application by using the application architecture considering specific application requirements [3]. With any increase in the number of application that is planned to develop in the product line using the common and variable artifacts, the number of available application architecture can grow extremely fast. In addition, in a software development lifecycle, it is crucial to make the software architecture understandable to all stakeholders involved in the project. One way to do this is making use of architectural views [4] that are defined for informing a stakeholder about a specific part of the software architecture considering the stakeholder's concerns [5]. In the context of SPLE, this means that there are a large number of architectural views in the whole product line since the number of available application architectures can grow fast and each of them generally has various architectural views because of the number of existing stakeholders in the project.

In a product line it is important to guarantee that, in terms of different architectural views, each and every one of the derived application architectures conforms to the reference architecture to ensure global consistency. However, in time, as the number of application architecture increases, it becomes impossible to follow traces of architectural conformance between the reference architecture and application architectures. In addition to this, due to evolution of the product line architecture and/or the application architecture, inconsistencies between these architectures might arise leading to an architecture drift which means that developed applications are not compatible with the product line and results sharply in decrease in general quality of software developed in the product line. In the literature several architecture conformance analysis approaches have been proposed but these have primarily focused on checking the inconsistencies between the architecture and code. Architecture conformance analysis within the

scope of SPLE has not got much attention.

1.2 Contribution

This work directly focuses on the architecture conformance analysis of reference architecture and application architectures that are derived from the reference architecture in a software product line. In this context, we firstly conducted a systematic literature review as a tertiary study in the field of software product line testing. Results of the tertiary study highlighted the lack of such an approach and a tool for architecture conformance analysis in software product lines. Then, we adopted the technique of reflexion modeling [6] and enhanced it according to the needs of SPLE. Reflexion modeling is very useful to demonstrate the results of comparison of two concepts; in our case, they are architectures.

For the process of architecture conformance analysis in SPLE, we developed a unique model-driven approach that includes the reflexion modeling technique. Furthermore, we added a tool support to our approach, which is a combination of programming language Java, Eclipse Epsilon Languages [7] and Apache Ant [8]. In addition, our systematic approach was applied to a small case study to demonstrate that it works as expected.

The contributions that this thesis makes can be listed briefly as follows:

- Systematic literature review as a tertiary study on software product line testing
- Reflexion modeling approach for SPLE
- Model-driven approach for architecture conformance analysis in SPLE
- Tool support for the proposed approach

1.3 Outline of the Thesis

This thesis is organized as follows: Chapter 2 provides the background information related to model-driven software development, software architecture design and general reflexion modeling technique. Chapter 3 presents the conducted systematic literature review in the area of software product line testing with its results. Chapter 4 discusses the reflexion modeling technique for SPLE and our model-driven approach. Chapter 5 presents an illustration of our approach for a concrete architecture framework. Chapter 6 describes details of the developed tool, its implementation and relation with the software architecture viewpoints, and the case study. Chapter 7 presents the related work. Lastly, Chapter 8 presents the conclusion.

Chapter 2

Background

This chapter presents the background on model-driven software development (MDSD) and architecture modeling, and reflexion modeling.

2.1 Model-Driven Software Development

As the name suggests, MDSD is about software development using models that are basically abstractions of a software system [9]. The methodology has been quite popular in the field of software engineering, which has its own developed open source tools in the market. The main idea of MDSD is that the code and documentation, i.e. design models, have equal value in the development phase; hence, via automatic transformations, the code can be generated from the models [9]. Key activities of the methodology are modeling, metamodeling and model transformations that can be model-to-model (M2M) and model-to-text (M2T) transformations.

In software engineering literature, the word modeling refers to the process of describing a software system under study. Therefore, a model is an asset that reflects possible answers of questions related to the software ignoring unnecessary details. There is a classification of models done by Mellor et al. [10] considering

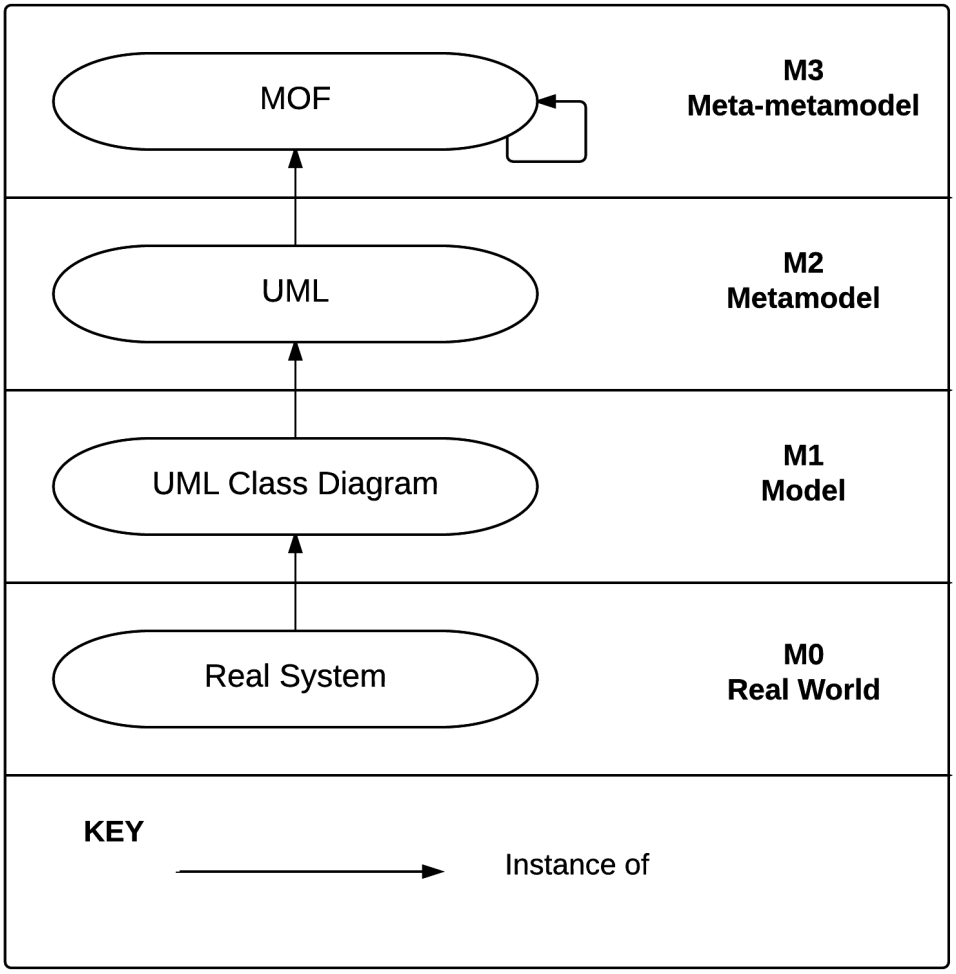

Figure 2.1: An example of four-layer OMG architecture

their level of precision. The first category considers models as a sketch that is used to communicate over ideas about a system, which are actually informal diagrams. The second category gives much more importance to models in a way that it considers them as blueprints that are, in fact, design models and provide wider and deeper information about the system. The last category lists models as executable, which means that they are executable artifacts as source codes. Likewise, this type of models can be automatically converted into another type of models or code. In MDSD technique, models are taken into consideration as executable, which gives way more precision than sketches and blueprints.

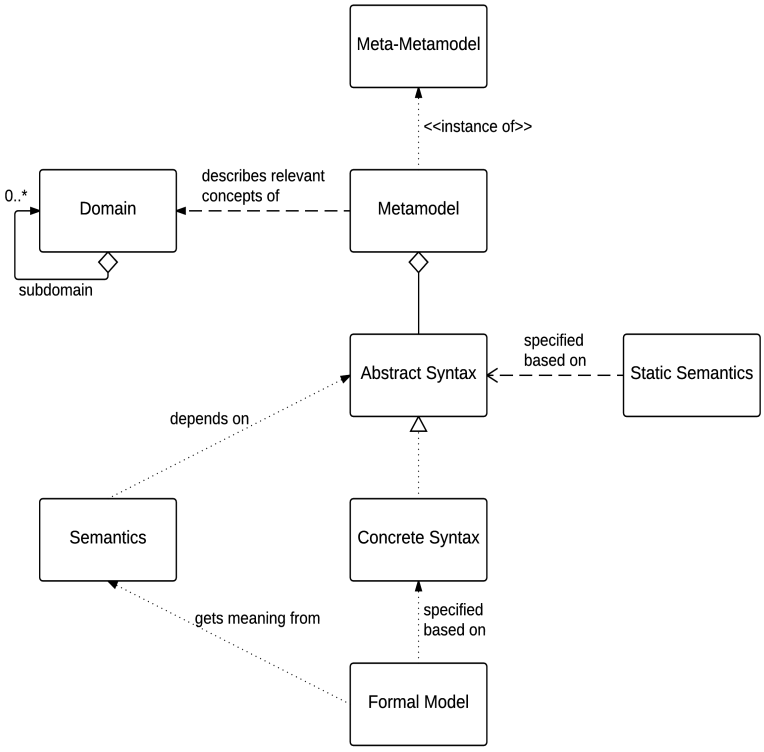

Figure 2.2: A conceptual model for metamodel concepts

In MDSD every model must conform to an upper level abstraction that defines rules the model must obey. This high level abstraction is called metamodel that is a model indeed and describes constructs of modeling language, their relationships, constraints and rules. Since a metamodel is a model, it has to conform to an upper level abstraction as well, which is called meta-metamodel. These concepts can be better explained by showing the four-layer OMG architecture [11] in which models are organized. In Figure 2.1 the structure with an example is demonstrated. From layer M0 to layer M3, the level of abstraction is increased and the real word concepts are defined in the layer M0. In the next layer, M1, ordinary user models are defined; in this case they are UML Class Diagrams. In layer M2, a metamodel for models that created in the layer M1 is defined; in this case this is called UML metamodel. In the highest layer, M3, a meta-metamodel is defined, which is a metamodel for metamodels created in the layer M2. In this example, in the layer M3, Meta-Object Facility (MOF) is given. In addition, in order to maintain this metamodeling structure, in the layer M3, the concept recursively defines itself.

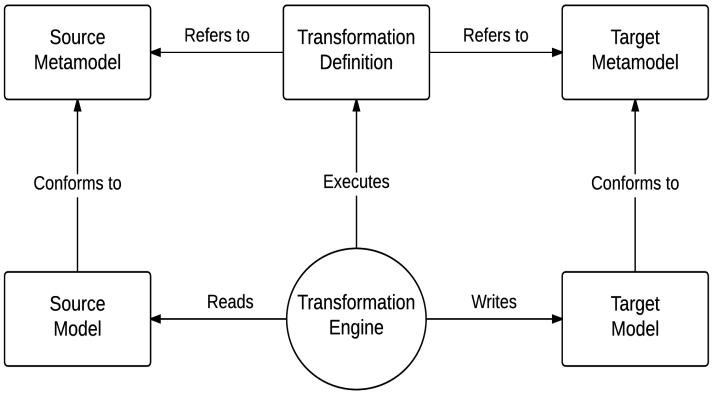

Figure 2.3: Model transformation process

In order to apply metamodeling process, one should consider fundamental elements of the metamodel, which are abstract syntax, concrete syntax, static semantics and semantics [9]. The abstract syntax is like a dictionary in which the language's vocabularies used to define its concepts exist. It also defines how these concepts can be combined to create models. The concrete syntax is a realization of the abstract syntax, which can be depicted as visually or textually. With a visual concrete syntax, diagrammatical version of a model is achieved; on the other hand, a textual concrete syntax provides a structural version of the model. Static semantics, i.e. well-formedness rules, express additional constraint rules that are hard to demonstrate on the abstract syntax. Lastly, semantics describe the actual meaning of concepts and relations defined in the abstract syntax. For a complete representation of metamodeling idea, please see Figure 2.2, which is self-explanatory.

Model transformation is a key activity in MDSD, which provides the functionality of M2M and M2T transformations. The former is used when a one type of model is required in the form of another type of model; for example, a generic test model is required to be converted into a JUnit test model. The latter is necessary when a design model is required to be converted into textual form; for instance, a JUnit test model can be translated into a Java test code via a M2T transformation.

In Figure 2.3, the basic elements and process of model transformation is demonstrated. In a transformation, the transformation engine, i.e. a software tool, reads an input source model that conforms to its given metamodel and considering the transformation definition, i.e. mapping rules, generates a target model that also conforms its given metamodel. The only difference between M2M and M2T transformations is that in the later there is no need for target metamodel. However, in the both transformations, the input and output models have to conform to their given metamodels. There are different developed tools for this purpose such as [12] and [7].

2.2 Software Architecture Design

A structure of a software system, which gathers software elements, relationships between them and their externally visible properties, is called software architecture of that software system [4]. Creating this architecture poses various issues to software architects, which has a crucial role in the development of large and complex software systems that have many stakeholders. The challenge is faced when it comes to make the software architecture understandable for all the stakeholders involved. In order to deal with this issue, the concept of architectural view is proposed. The goal of using architectural views is to provide a representation of a collection of software elements and their relations to a specific group of stakeholders considering their concerns [13]. Furthermore, the main advantage of an architectural view is its sustainability, which means that for each different potential stakeholder, there can be a separate architectural view that supports, by taking the stakeholder's needs into consideration, the modeling, interchange of ideas and reasoning related to the software architecture.

The standardization of architecture description is addressed in IEEE 1471 standard [1] in which the concept of viewpoint is introduced. An architectural view conforms to an architecture viewpoint that consists of design knowledge for creating and practicing a view [13]. The IEEE conceptual model for the architecture description can be seen in Figure 2.4 [1]. According to this, the stakeholders'

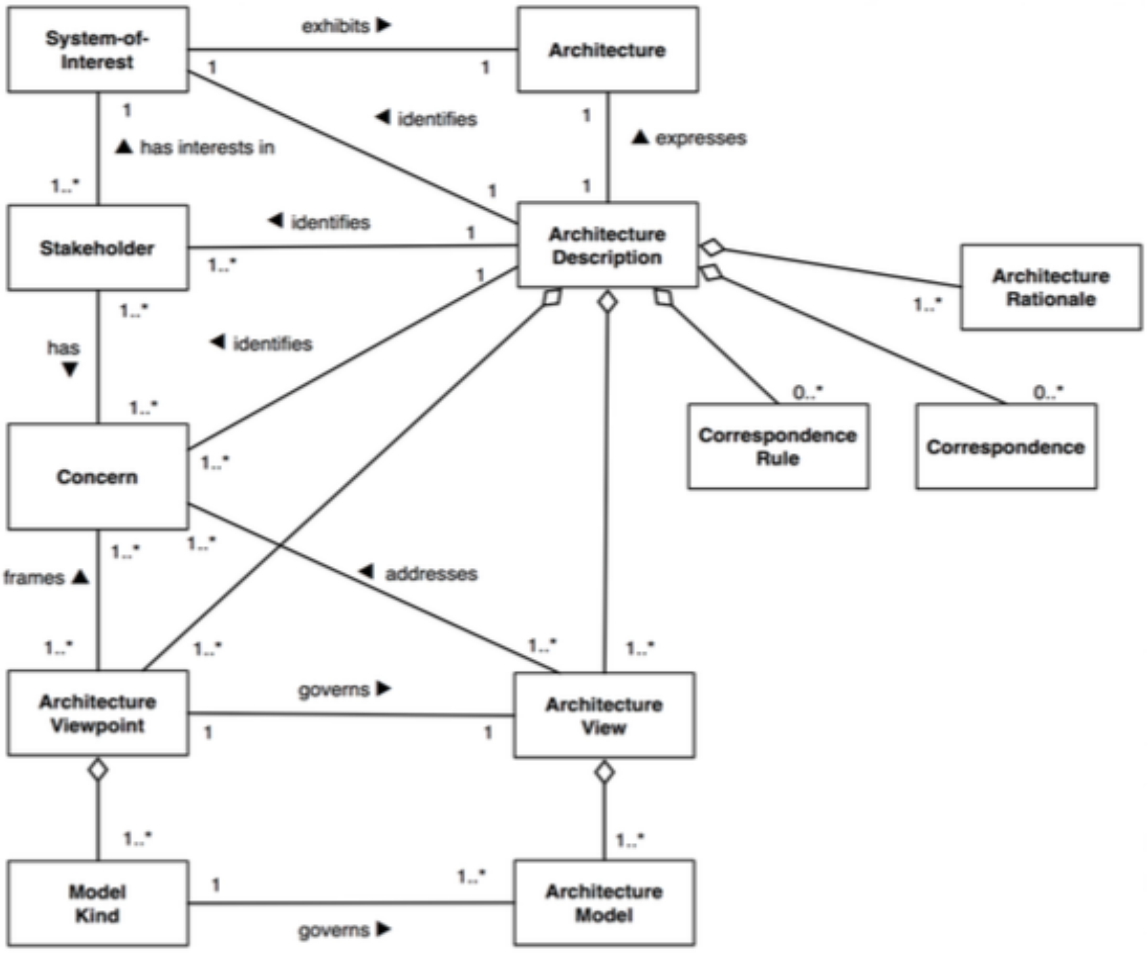

Figure 2.4: IEEE conceptual model for architecture description [1]

concerns' are addressed by architecture views that are governed by architecture viewpoints. In addition to this standardization, in software engineering literature, there are proposed software architecture frameworks, which documents and organizes the introduced architecture viewpoints, such as Kruchten's 4+1 approach, Rozanski and Woods viewpoint set, Siemens Four Views, and Views and Beyond (V&B) approach [4]. In this thesis, we study the viewpoint types given in the V&B approach that actually refers the viewpoint types as styles [4]. The information related to those viewpoints can be found in Section 4.3.

2.3 Reflexion Modeling

As software systems evolve through development and maintenance phases, their concrete artifacts such as design documents and source code need to be updated with respect to any changes made in that period so that the software artifacts can be coordinated and the end product can be reliable. Otherwise, a problem called drift between software artifacts might occur over time [6]. The drift issues generally appear between the description related to the software architecture and its implementation (this type of drift problem is called architectural drift [14]). In addition, in the context of product line engineering where architecture conformance is crucial, the drift might occur in architectural level due to inconsistencies between the product line architecture and an application architecture derived from it. The outcome of probable drift directly affects performance and general quality of the software artifact [15]. Therefore, it is highly important to identify and eliminate possible drift issues in order to guarantee that there are not any inconsistencies between software artifacts.

Design process of the software architecture, considering the whole software development period, is typically the starting point where the drift occurs. By taking this fact into account, many approaches for architecture conformance analysis/checking have been proposed in the software architecture literature. As it is underlined in [16] that the architecture conformance checking process can be done statically and dynamically, most of the proposed techniques appraise the case statically nevertheless. Two different studies, [16] and [17], that evaluate available static approaches for architecture conformance checking list the techniques as follows:

- Relation Conformance Rules (assessed only in [16])
- Component Access Rules (assessed only in [16])
- Dependency-Structure Matrices (assessed only in [17])
- Source Code Query Languages (assessed only in [17])

- Reflexion Models (assessed in both [16] and [17])

Relation conformance rules support a feature to indicate relations, which are allowed or forbidden, between two different components when checking their conformance [16]. Besides, the mapping operation of each conformance check is completed automatically. In the approach called component access rules, there are specified basic ports for components that can communicate with each other, i.e. with other components, via these ports [16]. In addition, the approach grants enabling only particular ports for component interaction, i.e. it provides encapsulation for components [16]. In dependency-structure matrices technique, software architectures are visualized by an effective abstraction in which a basic square matrix is used with rows and columns represent classes [17]. Moreover, expressiveness of the technique is limited to "can-use" and "cannot-use" constraints [17]. As a source code query language, the study [17] uses .QL language [17] that generates software architecture abstractions from code queries to compare.

Reflexion models technique was introduced by Murphy et al. [6] to assist software engineers to reason about their tasks by examining a generated high-level structural model of the task. The primary goal of Murphy et al. in developing the technique is to let engineers make use of the drift between the software architecture design and the implementation when they perform their tasks, instead of removing it. Basically what a reflexion model demonstrates is a compact representation of a software system's source model based on a high level model. The ease of using a reflexion model is the opportunity of selecting an appropriate view as a high level model for reasoning about the task. There are different projects in which the reflexion modeling is applied such as supporting design conformance analysis tasks and reverse engineering tasks of the Excel product [6].

The reflexion model approach has three main steps as follows (see Figure 2.5):

1. Defining a high level model of interest (HLM) that includes relations between its entities

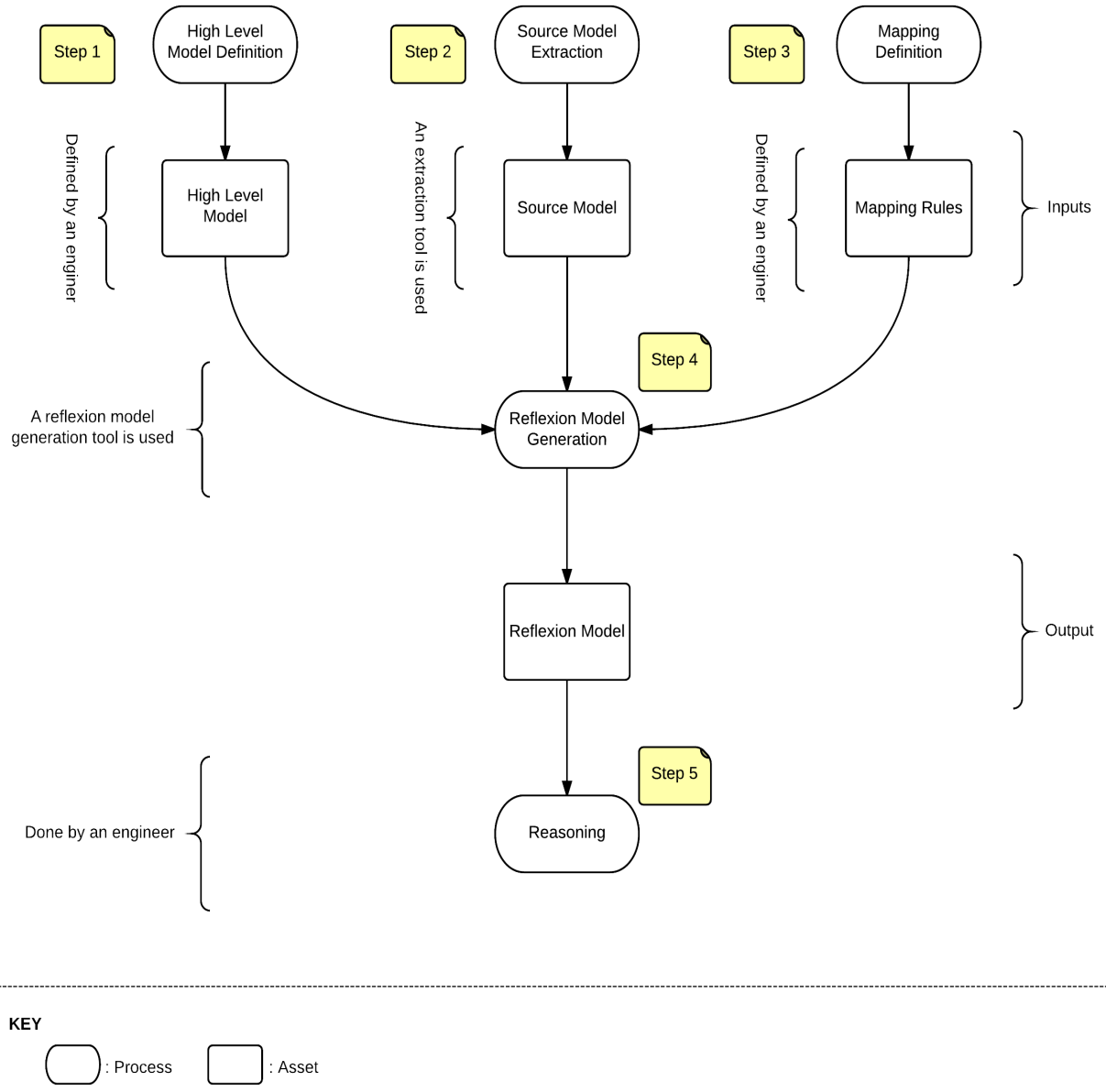

Figure 2.5: Steps of reflexion modeling approach

2. Extracting a source model (SM), which includes relations between its entities, from software artifact such as source code (generally done using a third-party tool)

3. Defining a mapping between the two models created in step 1 and step 2

The other two required complementary steps are as follows (see Figure 2.5):

4. Generating the reflexion model using a tool developed for reflexion model creation purpose

   (a) Inputs are the HLM, the SM and the mapping rules

   (b) Output is the reflexion model

5. Examining the reflexion model to reason about the task

When the reflexion model creation process, in the step 4 listed above, is initiated, the tool firstly compares relations exist in the HLM with relations exist in the SM to check whether each of relations exist in the HLM has a match in the SM with respect to the mapping rules that indicate which relation exist in the HLM is associated with which relation exist in the SM. After the comparison step, similarities, differences and missing parts between the models are computed and the reflexion model is generated mostly as a graphical representation (textual representation can also be possible depending on features of the used tool).

Murphy et al. name the relations, which are demonstrated in the resulted reflexion model, "convergence", "divergence" or "absence" [6]. If a relation that exists in the HLM also exists in the SM, then it is called convergence (i.e. expected [6]). If a relation that exists in the SM does not exist in the HLM, then it is called divergence (i.e. not expected [6]). If a relation that exists in the HLM does not exist in the SM, then it is called absence (i.e. expected but not found [6]).

The reflexion model graphically demonstrates convergence relation as a solid line, divergence relation as a dashed line, and absence relation as a dotted line.

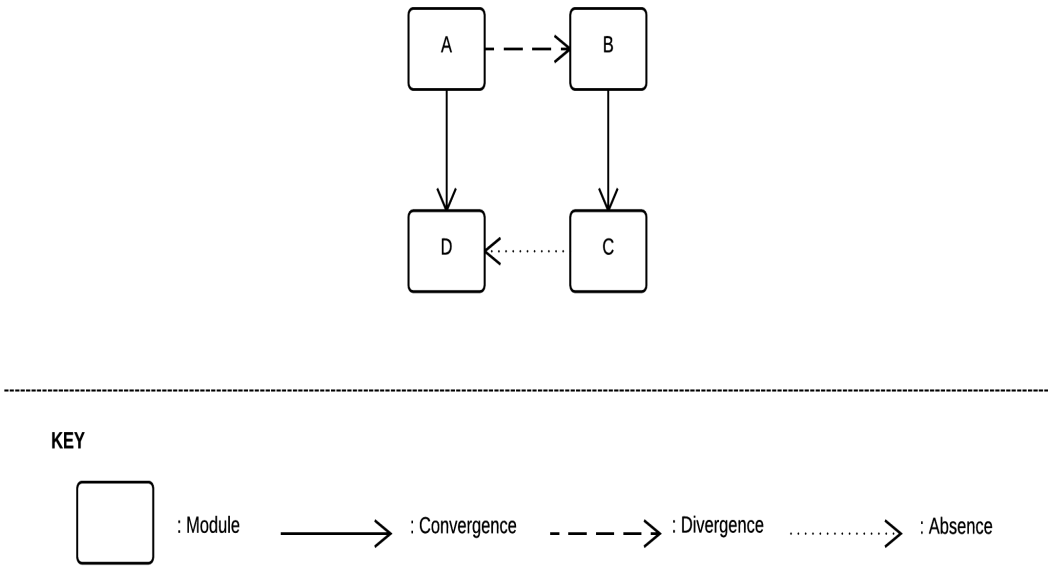

Figure 2.6: An example reflexion model

An example reflexion model is shown in Figure 2.6. It can be observed from the figure that the relation between module A and B exists in the SM although it does not exist in the HLM (divergence). The relation between module A and D, and the relation between module B and C exist in both the HLM and SM (convergence). The relation between module C and D exists in the HLM but does not exist in the SM (absence).

Finally, the reflexion modeling technique is an effective approach for software architecture conformance analysis and is easy to apply following some simple steps. Furthermore, the reflexion model is sufficient to reason about the task at hand. The explicit graphical representation of relation types in the reflexion model helps software engineers to identify the similarities, differences and missing parts between any two models readily.

Chapter 3

Systematic Literature Reviews in Software Product Line Testing: A Tertiary Study

Software product line (SPL) has been proven to be a building block of software development paradigms for large-scale software development organizations not only because it provides cost-effective and reliable software products through reuse but also it provides an opportunity of mass customization. The main idea of SPL is to guarantee the reuse is achieved through forward-looking, systematic and not ad-hoc way, in contrast with single-system engineering in which generally the software is first constructed and then reuse is taken into account. In this context, SPL consists of two different but tied activities: establishing a robust platform in which core assets are produced and deriving customer-specific applications from the platform [3]. This separation of two concerns pays off in terms of shortened development cycles and improved quality.

The importance of software testing is a well-known concept for single-system development for decades [18]. Likewise, it has become a crucial term when SPL is taken into consideration. Since SPL is a system that depends on the platform in which the common parts of software products are developed, it goes without

saying that the advantages of SPL are not realistic if the testing of each asset produced in the platform and each customized application constructed using the platform is done rigorously. A defect that is not revealed during the testing process of the core assets in the platform can be propagated to each customized software product of that SPL [19]. The cost of this situation might be very high in terms of money and time of fixing that fault. Therefore, the reliability of the platform is extremely important which should be tested thoroughly. However, exhaustive testing in SPL, which is a way to guarantee every single software artifact in the platform is going to work as expected, is not feasible due to number of inputs that can grow exponentially [20]. Furthermore, manual testing techniques [21] are not sufficient but can be complementary for SPL testing due to many input variables and complex development phases. In that respect, automated testing practices with tool support should be the first choice when products are tested.

SPL testing is not a mature concept but it is definitely a promising one. In that context, more research into SPL testing is escalating and expanding; hence, systematic literature review (SLR) and systematic mapping study (MS) are becoming available rapidly to shape the literature. In [22], SLR is defined as "a form of secondary study that uses a well-defined methodology to identify, analyze and interpret all available evidence related to a specific research question in a way that is unbiased and repeatable". In that sense, a SLR assists researchers to make connection between evidence and assessment of the research subject. A MS is a type of SLR that has research questions about a research topic at higher level of granularity compared to SLR and aims to form an overview of the research subject considering the type and amount of available studies with outcomes within it [23]. Another type of review, called tertiary review or tertiary study, is used to gather information and data from several SLRs in a specific domain. A tertiary study, which can be seen as a systematic review of SLRs [22], reflects the number of SLR available in a certain area, their topics and their overall quality. In addition, the methodology in a tertiary study and in a standard SLR is completely same [22].

This work presents a tertiary study using available SLRs in the field of SPL

testing. There is a lack of a tertiary study on SPL testing in the software engineering literature. The reason why we prepared a tertiary study instead of a SLR is that we aim to provide information about current status of research in this discipline and to identify open sub-sections for research and development using the already collected detailed and verified data in SLRs as a whole.

The remainder of the chapter is organized as follows: Section 3.1 presents the background of SPL, SPL testing and SLR (including MS and tertiary study). Section 3.2 gives the details of tertiary study protocol used in this study. Section 3.3 provides results of the tertiary study. Section 3.4 presents the discussion. Lastly, Section 3.5 concludes the chapter.

3.1 Background

3.1.1 Software Product Line

SPL is one of the promising software development paradigms, which has its own tools and techniques to produce a set of same family-based software products from a domain that consists of common core assets. There are also different names SPL is referred in Europe such as system family, product family or product population [24]. The reason why there are synonyms of SPL is that the communities of Europe and USA in this area worked independently until 1996 and the collections of jargon were already set [24]. In SPL the main goal of software development is to take advantage of reuse whenever it is possible. The opportunity of reuse is originated in building a domain in which fundamental software artifacts created. The actual software applications are developed using these core software artifacts and are customized based on their requirements.

SPL is composed of two different but related activities: domain engineering and application engineering [3]. The former is the part where a reusable platform is established; hence, the commonality and variability of the product line are defined and realized. The power of planned and steady reuse depends on traceability

links between the software assets, such as requirements, design, code and tests, produced in the platform [3]. The latter is the process of creating software applications by reusing core domain assets and making use of the variability. Likewise, it is the responsibility of application engineering that the binding of the variability is done correctly according to the applications' particular needs [3]. These two development processes have similar sub-activities within them that can be seen in Figure 3.1 [3]. In domain engineering there is an additional activity called business management, or product management [3], in which the product concerns related to market strategy and other economic parts are handled elaborately. The main aim of business management is the execution of product portfolio of the organization. Furthermore, this activity decides the scope of product line by explicitly identifying what product is in and what is out of the product line. In general, the other activities in domain engineering are domain analysis, domain design, domain implementation and domain testing. The reusable outputs of the activities in domain engineering are requirements, design, implementation (code), and test assets respectively. In application engineering, the artifacts produced in domain engineering are the inputs. Considering the variability model and reusing domain assets, in this development process the application specific assets are created and then individual products are realized (see Figure 3.1).

One of the remarkable points separates SPL from single-system development is systematic variability management. In this context, variation points are fundamental characteristics of a SPL, which represents points in design and code where single variants can be placed. A large number of software products can be developed when artifacts formed with variation points are generated with particular necessary variants chosen to use at each variation point. The effective tool of variability management is the variability model that is required to take advantage of a SPL significantly. This separate model is composed of variation points, variants, and their relationship [3]. It is constructed in domain engineering with necessary variants and realized in application engineering according to applications' needs.

In the process of specifying software products in a SPL the key role is played by features. They can be described as the distinct characteristic of a software

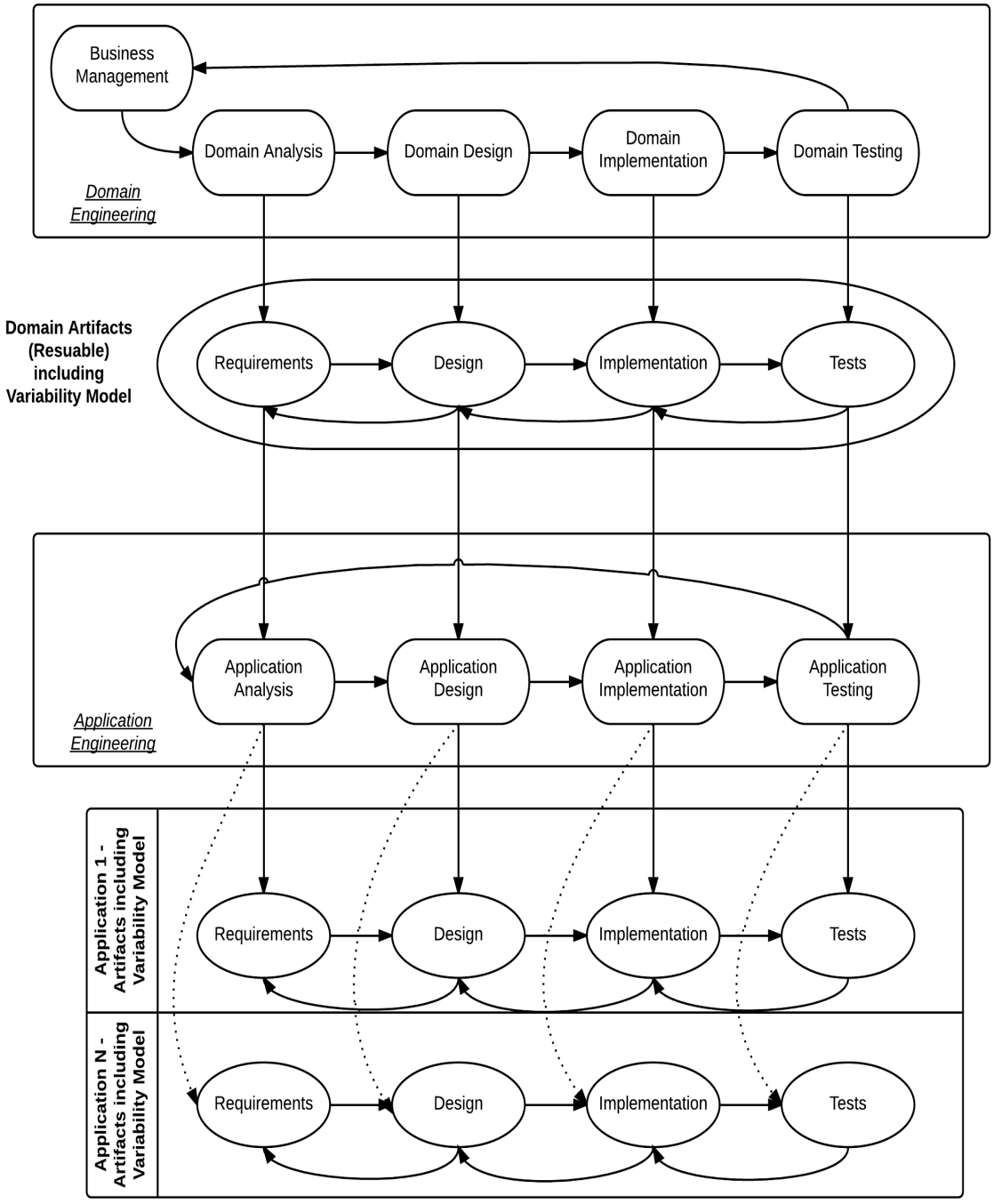

Figure 3.1: The framework of SPL

system specified by a collection of requirements [25]. Since a feature modeling provides a complete portrayal of all software products of a SPL in the light of their features, it has seen as the actual variability model [26]. The way feature models express the products in a SPL can be represented as a tree that adjusts the commonality and variability in its edges and the features in its nodes [27]. Then, following a path from the root to a leaf node constructs a product of that SPL.

The main motivation of launching a SPL in a company is related to its noteworthy advantages. These are listed in [3] and [28] as:

- Understanding of the domain is increased.
- Software product quality is improved.
- Customers' trust and satisfaction is enhanced.
- Systematic reuse of software artifact such as requirements, design and tests is advanced.
- Software quality control techniques are upgraded.
- Costs of software development and maintenance are reduced.
- Time to market is decreased.

On the other hand, concept of SPL comes with some drawbacks for companies such as [28]:

- Up-front investment cost is high, which is required to initiate the domain.
- Transition from a single-system development to a SPL development is hard to adapt in terms of staffs' mentality and organizational management.
- Inexperienced staff and lack of guidelines related to SPL affect the time to develop products in a negative way.

3.1.2 Software Product Line Testing

SPL testing activity is divided into two processes such as domain testing and application testing. Both are equally important and connected in terms of providing reusable test artifacts (from domain testing to application testing) and feedback about defects in test artifacts and request of test artifacts to be added into domain test artifacts (from application testing to domain testing). Furthermore, it is necessary that both testing activities must interact with each other in order to reduce general complexity of testing process in the SPL.

The goal of domain testing is to guarantee that all domain artifacts are free of defects, i.e. checking whether output of domain analysis, domain design and domain implementation has a bug, and to produce reusable test artifacts for application testing. It can be said that domain testing activities are similar to ones in single-system software testing; however, it has to cope with variability [3].

The aim of application testing is to test each product line applications using reusable domain test assets to see whether or not they have any defects. However, this is not efficient to verify the absence of faults. Therefore, it is required to test each software product thoroughly.

Detailed and extensive testing of both domain test artifacts and product line applications are generally infeasible and problematic since there is a huge number of an input combination due to the variability in features. In addition, the effectiveness of current testing tools is not enough to handle the situation of having a large number of test inputs.

3.1.3 Systematic Literature Review

A SLR is a method to examine, evaluate and synthesize all available studies related to specific questions, a particular field, or topic of interest in a way that is systematic, trustworthy and rigorous [29]. The goal of performing SLR is to gain information about the existing evidence for a particular area of interest, to

point any empty spots in that area of research in order to suggest for further investigation and to create a base for new research activities [22]. In addition, A SLR is considered as a secondary study that consists of preliminary studies about the topic of interest that is investigated. There is also another type of SLR, which is also a secondary study, called MS or "scoping study" [22]. The aim of performing a MS is to provide a wide overview of a phenomenon of interest. The main difference between a MS and a SLR is that the former generally has broader research questions and may investigate manifold research questions. Yet another type of SLR is called tertiary study or tertiary review that complements SLRs and known as the review of secondary studies [30]. Tertiary study is available to conduct when there is a number of SLR and MS exist already in the literature of topic of interest for finding answers of wider research questions. It can be said that a tertiary review needs less resources than a SLR but its resources, in this case SLRs, should be in a high quality [22].

The origin of SLR is connected to a methodology, called evidence-based research, which initially evolved from the area of medicine. This is a kind of interesting evolution since software engineering and medicine are two different fields. The key reason to adapt the core idea of evidence-based medicine to software engineering is the success of evidence-based medicine. In terms of software engineering this new methodology is called evidence-based software engineering (EBSE) that includes SLRs, MSs and tertiary studies as an important part [31]. In [32], the goal of EBSE is given as " to provide the means by which current best evidence from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software". It can be inferred that EBSE provides a standard to practitioners in terms of a guideline that can be used to guarantee their research is followed by the requirements of industry and partner groups.

3.2 The Tertiary Study Protocol

This study is conducted as a tertiary study to identify and evaluate the current evidence regarding SPL testing. As we prepare our tertiary study, we follow the guidelines proposed by Kitchenham et al. [22][30]. Since a tertiary study is a review of secondary studies, the methodology of a tertiary study is exactly same as a SLR. The review protocol is composed of identification of research questions, definition of search strategy with search strings, definition of inclusion/exclusion criteria, definition of quality assessment method, design of data extraction form with process of performing pilot data extraction, and definition of data synthesis method (see Figure 3.2).

As an initial step of conducting a tertiary study, we firstly developed a review protocol that describes fundamental methods that are used to perform a tertiary study. The advantage of using a pre-defined protocol decreases the researcher bias. Our review protocol is demonstrated in Figure 3.2.

The first step of the tertiary study protocol is to specify research questions related to the objectives of this tertiary review (Section 3.2.1). The next step is to define search strategy including search scope. In the search strategy step we designed the search strings that were formed after performing pilot searches that can be used to infer reasonable conclusion (Section 3.2.2). In this point, quality of a search string is crucial since a well-planned search string fetches appropriate and sound outcomes, which is also important in terms of its degree of accuracy. The search scope denotes the time span and the venues we looked at in order to find the essential resource of this study, in this case the secondary studies. After the search strategy was defined, we specified the study selection criteria that were used to determine which studies are included in or excluded from the tertiary study (Section 3.2.3). The secondary studies were screened at all phases on the basis of inclusion and exclusion criteria. Moreover, we took advantage of peer reviewing throughout the study selection process. As a next step, we performed quality assessment process in which the secondary studies resulted from the search process were investigated based on quality assessment checklists

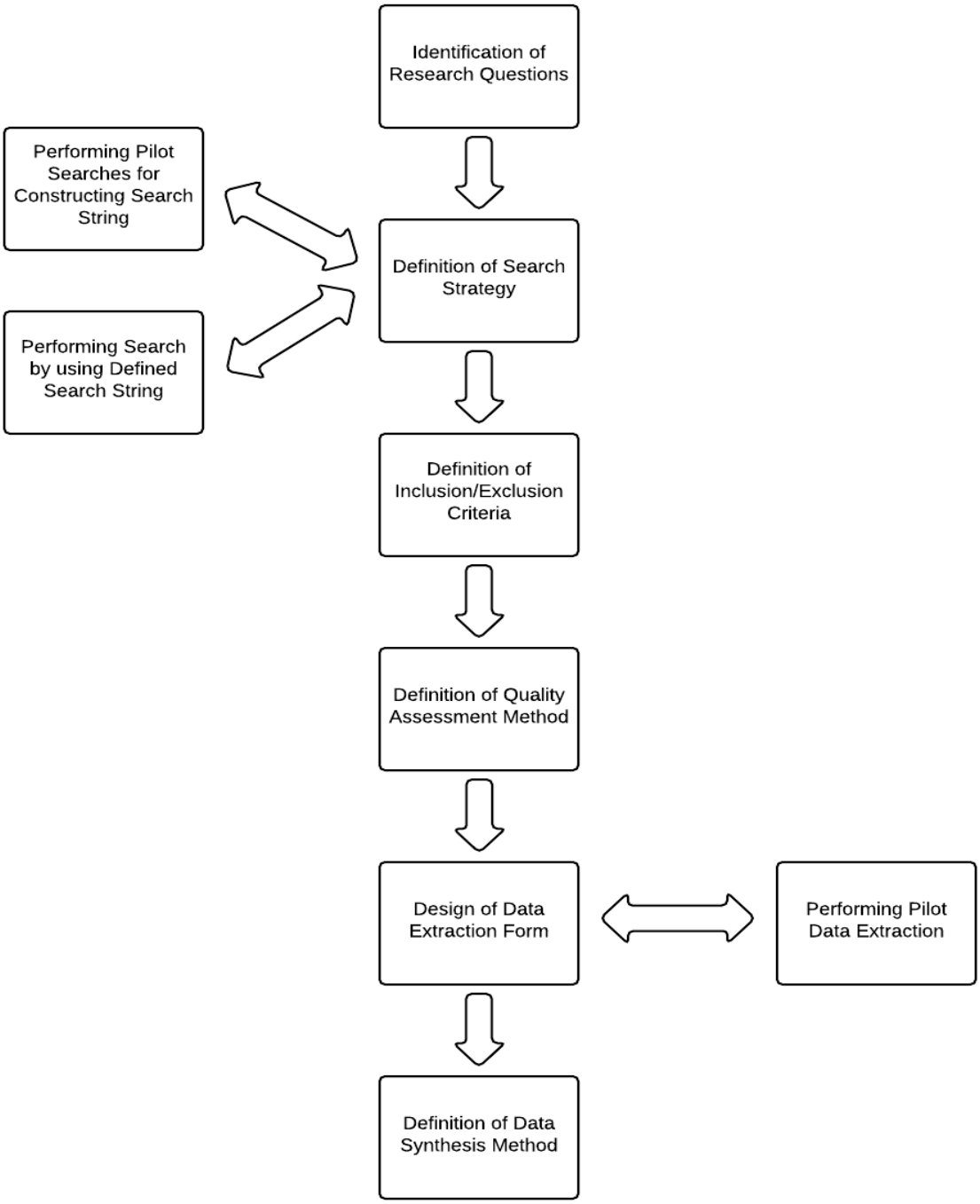

Figure 3.2: The tertiary study review protocol

and procedures (Section 3.2.4). The process followed with development of data extraction strategy, as the final set of secondary studies is fixed, that defines how the information required from each study is obtained (Section 3.2.5). For this purpose we developed a data extraction form that was defined after a pilot study. In the last step of the protocol data synthesis method definition takes place in which we present the extracted data and associated results.

3.2.1 Research Questions

The core building block of any type of systematic review is its clear-cut, explicit and deductive research questions since they are the initiator that drives subsequent parts of the systematic review. Therefore, it is extremely significant to ask right questions to obtain relevant findings completely. While we were trying to develop precise research questions to find accurate findings, at the same time we had to consider that they should be wide enough since a tertiary study asks wider research questions compared to a SLR and a MS. Therefore, our research questions cover not too much deep points of the field of SPL testing. The following is the list of our research questions:

RQ1: What SLR studies have been published in the area of SPL testing?

RQ2: What SPL testing research topics are addressed?

RQ3: What research questions are being investigated?

It is important to note that in RQ1 the term "SLR" stands for both conventional SLRs and MSs as secondary studies.

3.2.2 Search Strategy

The main goal of performing a tertiary study is to find as many secondary studies relating to the research questions as possible using a well-planned search strategy.

In this part of the study we define our search strategy by explaining search scope, adopted search method and search string.

Scope

As a search scope we give importance to two dimensions that are publication venues and publication period. The publication venues consist of journals, conferences and workshops. With respect to the publication period our search scope includes the papers that are published between the period of 2004 and March 2015. The search was conducted in September 2014 and replicated in March 2015.

Search Method

As a search method we used a systematic and evidence-based approach that gives an optimal search strategy rather than ad-hoc techniques, which is given in [33]. This method is called as quasi-gold standard (QGS) that uses two metrics such as sensitivity and precision to evaluate search performance. The former is considered as, for a given subject, the proportion of relevant studies fetched for that subject and the latter is defined as the proportion of fetched studies that are relevant studies. The gold standard refers to the known collection of identified studies in a set with respect to the definition of research questions given in a SLR [33]. Furthermore, if the level of sensitivity of the search strategy is high, then it is expected to get most of the studies in the gold standard, however, with the chance of getting a lot unwanted works. If the degree of precision of the search strategy is high, then the expectation is to get less irrelevant studies, however, with the chance of missing many studies in the gold standard. The difference between the concepts of QGS and gold standard is that the former has two more constraints in search process such as the venues where relevant studies are published and the period to be searched.

According to [33], it is important to take following points into consideration in order to define effective search strategy for study/evidence searches in a SLR: the approach to be used in search process (manual, automated or both), the venues or

databases to be used in search process and the field of article to be searched, the subject and evidence type to be searched and the search strings fed into search engines, and the time when the search is carried out and the time span to be searched. Likewise, the QGS consists both manual search and automated search to provide an optimum search strategy [33]. The databases used in automated search and the venues used in manual search in our search method can be found in Appendix-A. It is important to underline that some of the websites of journals, conferences and workshops used as venues in the manual search are not available; hence, we used the website dblp - computer science bibliography [34] to access the papers.

Search String

The search string is used in automated search, which is constructed after performing a number of pilot searches to retrieve relevant studies as much as possible. Since each electronic search engines provides different features, for each search engine, we define different search strings that are semantically equivalent. Each search string is applied to title, abstract and keywords of the papers. Complex search queries are created with OR and AND operators. The following represents the search string that is defined for ScienceDirect database:

> tak(("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study"))

The complete set of search strings used in electronic databases is given in Appendix-B. Since some search engines do not support long queries or multiple-part searches at the same time such as searches in title, abstract and keywords

Table 3.1: Total number of studies obtained/reviewed after search process and study selection

Source	**Number of Studies Gathered From Search Process**	**Number of Studies Reviewed After Search Process**	**Number of Studies After Exclusion Criteria Applied**
ACM Digital Library	5	5	3
CiteSeerX	0	0	0
Google Scholar	3800	50	0
IEEE Explore	2	2	0
ScienceDirect	4	4	0
Scopus	32	32	0
Springer Link	1134	100	1
ISI Web of Knowledge	8	8	0
Wiley InterScience Online Library	26	26	0
Manual Search	3	3	0
Total	5014	230	4

of the papers together, we split the search strings according to their defined rules (See Appendix-B).

The number of studies gathered from both automated and manual search processes is given in the second column of Table 3.1. In the first stage of search process we found 5014 papers. The search engines Google Scholar and Springer Link provided a huge number of outputs. Since these two search engines list their results by considering their strong relevance with the search query, we only looked first 5 pages of the result. The following pages fully composed of unnecessary and irrelevant studies. Therefore, the number of reviewed papers found in Google Scholar and Springer Link electronic databases was 50 and 100, respectively. Likewise, different search engines can find same papers simultaneously or the paper may have different versions within an electronic database. We considered this type of situations when we developed our exclusion criteria (see Section 3.2.3). After applying them, the total number of studies to analyze was reduced to 4.

3.2.3 Inclusion and Exclusion Criteria

Inclusion and exclusion criteria are two important concepts to consider when any type of SLR is conducted since they can be used to decide which studies are in or out of the scope of SLR. Moreover, exclusion criteria are the tool to reduce a large number of outputs that is resulted from the automated search process to a reasonable number. Our inclusion (IC) and exclusion criteria (EC) are listed in the following separately as:

- IC1: Studies that have full text available are included.
- IC2: Studies that are written in English are included.
- IC3: If there are many different versions of a study, the most current version is included.
- EC1: Studies that do not relate SPL testing field are excluded.
- EC2: Studies that are not any type of SLR conducted by following valid guidelines such as [22] are excluded.
- EC3: Duplicate publications obtained from different search engines are excluded.
- EC4: Surveys, Masters and PhD studies are excluded.
- EC5: Studies that are solely composed of tool support, i.e. tool description, for SPL testing are excluded.

3.2.4 Quality Assessment

Defining inclusion and exclusion criteria is not enough to guarantee the quality of secondary studies resulted from the search process is close to standards or above them while conducting a tertiary study. It is also necessary to assess the quality of secondary studies to detail available inclusion and exclusion criteria, to

point out the significance of secondary studies when they are being synthesized, to provide a way to interpret the findings, and to guide recommendations for further research.

The main challenge of this step is that there is not, by common consent, any definition of word "study quality". For any secondary study, in order to prepare quality assessment step, one can follow the guideline [22] in which the definition of word "study quality" is depended on quality instruments that are actually checklists of elements that need to be considered for each work. In [22], both summary quality checklist for quantitative studies and checklist for qualitative studies are described and given in detail. There is also a version of [22] for tertiary studies to be a guideline to prepare quality assessment process that is [35] by the same author of [22]. In our tertiary study as some other tertiary studies such as [29] and [36], we used quality assessment questions described in [35] that can be seen in Table 3.2. As it is highlighted in [22], by assigning quality items within a checklist to numerical scales, we can obtain numerical assessments of study quality to rank secondary studies according to their overall quality score. In addition, we use the three-point scale and assign scores (yes (Y)=1, partly (P)=0.5, no (N)=0) to the each criterion. The obtained secondary studies after search process and exclusion criteria applied are listed in Appendix-C and the result of quality assessment process can be seen in Appendix-D.

The four questions given in Table 3.2 were scored in [35] as follows:

- "QA1: If the inclusion criteria are explicitly defined in the paper, we give Y. If they are given implicitly, we give P. If they are not defined, we give N.
- QA2: If the authors searched at least 4 electronic databases and have additional search strategies such as manual searching, or identified and referenced all journals addressing the topic of interest, we give Y. If they looked at 3 or 4 electronic databases without additional search strategies, or searched a defined but restricted set of journals and conference proceedings, we give P. If the authors have searched up to 2 electronic databases or an extremely restricted set of journals, we give N.

Table 3.2: Quality checklist

Reference	Questions
QA1	Are review's inclusion and exclusion criteria described and appropriate?
QA2	Is the literature search likely to have covered all relevant studies?
QA3	Did the reviewers assess the quality/validity of the included studies?
QA4	Were the basic data/studies adequately described?

- QA3: If the authors have explicitly defined quality criteria and extracted them from each primary study, we give Y. If the research question involves quality issues that are addressed by the study, we give P. If no explicit quality assessment of individual papers has been attempted or quality data has been extracted but not used, we give N.

- QA4: If information is presented about each paper so that the data summaries can clearly be traced to relevant papers, we give Y. If there is only summary information is presented about individual papers, we give P. If the results of the individual studies are not given, we give N."

3.2.5 Data Extraction

The aim of data extraction process is to design data extraction forms to store the data researchers get from the secondary studies [22]. Likewise, data extraction forms are necessary to obtain all information needed to point out the research questions and the study quality criteria. In our data extraction forms, we highlighted fundamental aspects of studies such as their titles, authors' names, publication venues, and their source as well as their important parts that address our research questions such as publication type (RQ1), main SPL testing topic (RQ2), and their research questions (RQ3). Data extraction process was performed independently by two researches to ensure data extraction consistency. The complete data extraction form can be seen in Appendix-E, which is based on the data collection list for tertiary studies in the guideline [22].

3.2.6 Data Synthesis

Extracted data, in this step, is collated and summarized in an appropriate way to answer research questions of a tertiary study. As it is stated in [22], we tabulated the data to demonstrate the fundamental information about each study. After the tabulating process, we reviewed the table to answer our research questions. Regarding the research questions, the data synthesis stage proposed a way to answer them as follows:

- *RQ1: What SLR studies have been published in the area of SPL testing?*
  - This was addressed by looking at the type of SLR papers (SLR or MS).
- *RQ2: What SPL testing research topics are addressed?*
  - This was addressed by thoroughly reading all papers.
- *RQ3: What research questions are being investigated?*
  - This was addressed by analyzing all research questions in each SLR study.

3.3 Results

3.3.1 Data Extraction Results

The data we obtained from the secondary studies is presented in Table 3.3, Table 3.4 and Table 3.5, and additionally with a bulleted list given at the end of this sub-section. The summary data of the studies is given in Section 3.3.2 and the detailed quality assessment scores of the studies are demonstrated in Appendix - D.

The four reviewed secondary studies are linked with the paper references "A, B, C and D" as it is shown in Appendix-C, Table 3.3, Table 3.4 and Table 3.5.

Table 3.3: Extracted data of each study - 1

Paper Reference	Title	Year
A	B.P. Lamancha, M. Polo, M. Piattini. Systematic Review on Software Product Line Testing	2013
B	E. Engstrom, P. Runeson. Software Product Line Testing - A Systematic Mapping Study	2011
C	I.d.C. Machado, J.D. McGregor, Y.C. Cavalcanti, E.S.d. Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review	2014
D	P.A.d.M.S. Neto, I.d.C. Machado, J.D. McGregor, E.S.d. Almeida, S.R.d.L. Meira. A Systematic Mapping Study of Software Product Line Testing	2011

In Table 3.3, the title of the studies with their authors and their publication year are listed.

In Table 3.4, we listed the repository name, i.e. name of the digital library, which the studies were downloaded from, the source type that can either be a conference or a journal, the publication venue where the study was published, and the type of the secondary study.

It can be said that each of the studies A, B, C and D was prepared using a guideline. In addition, except study A, the studies have both manual and digital searches together. Although study B's literature search activity includes manual and digital searches, its quality assessment score of literature search (QA2) is "N", i.e. zero, since study B's authors searched only 2 electronic databases (Google Scholar and ISI Web of Knowledge) and extremely restricted set of workshops and conferences (only SPLiT and SPLC, respectively).

In order to define "Scope" and "Main SPL Testing Topic" fields given in the below bulleted list, we consulted the guidelines [35] and [39], respectively. For the former, the [35] suggests that a study's scope is "Specific Research Question" if it is posed detailed technical questions regarding the field of interest or is "Research Trends" if it is mainly related to trends of a specific software engineering field.

Table 3.4: Extracted data of each study - 2

Paper Reference	**Repository**	**Source**	**Publication Venue(s)**	**Type**
A	Springer Link	Conference	ICSOFT 2010	SLR
B	ACM Digital Library	Journal	Information and Software Technology	MS
C	ACM Digital Library	Journal	Information and Software Technology	SLR
D	ACM Digital Library	Journal	Information and Software Technology	MS

Table 3.5: Extracted data of each study - 3

Paper Reference	**Study Guideline(s)**	**Number of Primary Studies Included**	**Literature Search Type(s)**	**Quality Assessment Score**
A	[37]	37	Digital	2.5
B	[23]	64	Manual + Digital	2
C	[22]	49	Manual + Digital	3.5
D	[38]	45	Manual + Digital	3.5

In our case, the studies A, B and C provide research questions that are interested in the trends of SPL testing field; on the other hand, the study D directly asks technical research questions to find out technical aspects of SPL testing area. For the latter, it can be underlined that the authors of [39] divide concept of Software Testing into 5 main sub-sections that are Software Testing Fundamentals that have testability issues, Test Levels such as unit testing, integration testing, etc., Test Techniques such as code-based testing techniques, fault-based testing techniques, etc., Test Related Measures that cope with test cost and effort issues, and Test Process that is corresponding to test activities. Although these sub-sections are given for Software Testing in general, they are still applicable for SPL testing as well. However, the classification that [39] provides does not include some crucial aspects of SPL testing such as dealing with variability and creating reusable components. Therefore, the extracted data from the secondary studies is categorized more detailed in Section 3.3.2 where overview of the studies provided.

The extracted data in terms of scope, main SPL testing topic, research question(s) and research objective(s) for each paper are given as follows:

- *Study A - Lamancha et al.*
  - **Scope:** Research Trends
  - **Main SPL Testing Topic:** Test Levels, Test Process, Test Related Measures, Test Techniques
  - **Research Question(s):** "Which initiatives have been carried out to deal with testing in the Software Product Lines?"
  - **Research Objective(s):** "To analyze the existing approaches to testing in software product lines, discussing the significant issues related to this area of knowledge and providing an up-to-date state of the art, which can serve as a basis for innovative research activities."
- *Study B - Engstrom and Runeson*
  - **Scope:** Research Trends

- **Main SPL Testing Topic:** Test Process, Test Levels, Test Techniques, Software Testing Fundamentals, Test Related Measures
- **Research Question(s):** "Which challenges for testing software product lines have been identified?", "In which fora is research on software product line testing published?", "Which topics for testing product lines have been investigated and to what extent?", "What types of research are represented and to what extent?"
- **Research Objective(s):** "It aims at surveying existing research on SPL testing in order to identify useful approaches and needs for future research."

- *Study C - Machado et al.*
  - **Scope:** Research Trends
  - **Main SPL Testing Topic:** Test Techniques
  - **Research Question(s):** "What SPL testing strategies are available to handle the selection of products to test?", "What SPL testing strategies are available to deal with the test of end-product functionalities?", "What is the strength of the evidence in support of these proposed SPL testing strategies?", "What are the implications of these findings for the software industry and the research community?"
  - **Research Objective(s):** "To identify testing strategies that have the potential to achieve economies of scope and scale in the testing activities, and to provide a synthesis of available research on SPL testing strategies, to be applied towards reaching higher defect detection rates and reduced quality assurance effort."

- *Study D - Neto et al.*
  - **Scope:** Specific Research Question
  - **Main SPL Testing Topic:** Test Techniques, Test Levels, Test Related Measures
  - **Research Question(s):** "Which testing strategies are adopted by the SPL Testing approaches?", "What are the existing static and dynamic

analysis techniques applied to the SPL context?", "Which testing levels commonly applicable in single-systems development are also used in the SPL approaches?", "How do the product line approaches handle regression testing along software product line life cycle?", "How do the SPL approaches deal with tests of non-functional requirements?", "How do the testing approaches in an SPL organization handle commonality and variability?", "How do variant binding times affect SPL testability?", "How do the SPL approaches deal with test effort reduction?", "Do the approaches define any measures to evaluate the testing activities?"

- **Research Objective(s):** "Investigate state-of-the-art testing practices, synthesize available evidence, and identifying gaps between required techniques and existing approaches, available in the literature."

3.3.2 Overview of The Reviewed SLRs

Study A - Lamancha et al.

This study presents a systematic literature review of testing in software product lines, which mainly aims to investigate the current approaches for dealing with testing in SPL, highlighting the important issues related to this field, and creating a trustworthy infrastructure for further research activities. In addition, it includes a separate work that is out of the set of primary studies included in the SLR, which is provided by Bertolino [40] that is a sort of a roadmap to understand the gains, difficulties, and goals in software testing research. Likewise, Lamancha et al. match the primary studies at hand with the goals given in the [40].

The study categorizes the data extracted from each primary study reviewed in the SLR in unit testing (UT), integration testing (IT), functional testing (FT), SPL architecture testing (AT), embedded systems testing (ET), testing process (TP), testing effort in SPL (TE), and testing product generation (PG). Lamancha

et al. state that most of the works deal with testing levels in SPL testing topic and specifically a remarkable interest is shown in FT and variable testing with UML models and use cases. Moreover, there exist a few proposals in terms of test automation, which are typically prototypes. It is also indicated by Lamancha et al. that there is not enough effort to put theoretical proposals into practice since the conducted experiments and case studies documented are scarce. Furthermore, they emphasize that it can be felt the lack of a testing process designed for SPL because the existing studies proposed for this interest are guidelines. Another deficiency of existing SPL testing literature is composed of test oracle definition, testing planning, management, and cost estimation topics.

The difficulties or "challenges" of software testing, which are also applicable to SPL testing field, given within a goal or "dream" of software testing in [40] as follows:

- **Goal** - Universal Test Theory: Having a common, compatible, and thorough framework which testers can consult when they need to understand the strengths and limitations of available test techniques, and can make use of it in choosing the suitable one, considering the present conditions.
  - **Difficulty** - Explicit Test Hypotheses: When selecting a test set as a delegate of some likely executions, the test activities should be explicit.
  - **Difficulty** - Test Effectiveness: In order to understand the sets of errors, one should derive analytical or empirical evidence of the effectiveness of the test selection criteria in revealing errors.
  - **Difficulty** - Compositional Testing: It is necessary to comprehend the way we reuse test results of the unit testing, the conclusions that can be drawn from the system caused by the composition, and which extra test cases should be run on the integration.
  - **Difficulty** - Empirical Body of Evidence: In order to create a base for producing and evolving a theory for testing, one should generate the necessary empirical body of evidence.

- **Goal** - Test-based Modeling: Considering a way of best possibly building the software model in order to test the software afterward effectively.
  - **Difficulty** - Model-based Testing: It is significant to realize the way we can combine various modeling styles and the need for approaches to integrate model-based criteria with other techniques.
  - **Difficulty** - Anti-model-based Testing: Unlike model-based testing, it is needed to make an effort to generate innovative styles of testing that is related directly to the analysis of program executions.
  - **Difficulty** - Test Oracles: There is a need for more efficient and automated test oracles.
- **Goal** - 100% Automatic Testing: Creating an efficient, robust and rapid integrated test environment that by itself can automatically handle controlling it, producing required driver or stub codes, generating proper test suites, running them and finally issuing a test report.
  - **Difficulty** - Test Input Generation: Basically, it is needed to automatically generate test inputs.
  - **Difficulty** - Domain-specific Test Approaches: It is needed to create domain-specific techniques and tools to advance test automation.
  - **Difficulty** - On-line Testing: Using dynamic analysis and self-test methods, we can monitor a system's behavior in real life operation.
- **Goal** - Efficacy-maximized Test Engineering: Having a valid and feasible test process providing the maximum efficacy through the all required subactivities of software testing.
  - **Difficulty** - Controlling Evolution: It is needed to have new approaches to control regression testing process when dealing with large composite systems.
  - **Difficulty** - Leveraging User Population and Resources: Enhance inhouse activities of testing and maintenance by using collected data from the area.

- **Difficulty** - Testing Patterns: Testing patterns are highly attractive in the field, which offer effective solutions to recurring testing problems.
- **Difficulty** - Understanding the Costs of Testing: Novel methods are needed to employ testing techniques efficiently considering a fixed testing budget.
- **Difficulty** - Education of Software Testers: The basic idea of testing, limitations and opportunities introduced by the existing methods should be taught to testers.

Lamancha et al., referring to [40], states that the topics of model-based testing and test input generation are the fundamentals for SPL testing field. They also add that other challenges are not yet faced by the researchers of SPL testing area. Lastly, they mention their work that is applying model-driven techniques to generate test cases in software product lines. In their proposal, test code is generated from design models automatically using combination of QVT transformations [41] and MOFToText transformation [42].

Study B - Engstrom and Runeson

In this work, Engstrom and Runeson provide a systematic mapping study that targets to demonstrate an overview of current research on product line testing. The study consists of several research questions to find out existing challenges for SPL testing, to point out the fora in which research on SPL testing published, to realize topics of SPL testing addressed and their scope, and to list types of research conducted in this field.

Engstrom and Runeson divide challenges related to SPL testing into 3 concerns: coping with huge number of tests, coordinating the effort required to test both reusable components and concrete products, and dealing with variability. Regarding first concern, Engstrom and Runeson highlight that, for a complete test to be done in a product line considering both domain artifacts and concrete products, one should test all probable uses of each asset and possible product variants. Since the number of variants is prone to grow exponentially with the

number of variation points, that highlight remains unreal to perform. Likewise, they state that the more developed actual products in SPL, the more need for system testing to run. The main problem in this point is that there can be many redundant test cases to execute due to a huge number of variants. As a matter of fact, it is possible to get rid of such redundant tests by reusing test artifacts since the actual products are nearly composed of same set of specifications in the product line. With regard to the second concern, Engstrom and Runeson indicate that it is a must to balance the effort spent on testing reusable domain artifacts and actual products. Moreover, it is still not clear that there is an advantage of primarily testing of reusable components to reduce testing obligations for each product, as Engstrom and Runeson underline. Lastly, they add that since the development process in SPL is divided into two parts so that testing process is done by different groups, the planning of whole development process, especially testing activities, should be done carefully. In terms of the third concern the authors underline that it is not obvious how testing of distinct types of variation points should be handled since the types of interdependencies they include are varied. Besides, it is necessary to verify incorrect bindings of variation points do not exist since all application engineering products should be consisted of their exact features, not of any features that exist there by mistake.

The primary studies, which are reviewed in the study B, are categorized by three different aspects: main focus that the research targets, contribution type, and research character. Then, each category is divided into some sub-sections. The class of research focus consists of 6 sub-sections such as test organization and process, test management, testability, system and acceptance testing, integration testing, unit testing, and automation. The category of contribution type is composed of 5 sub-sections that are tool, method, model, metric, and open items. Last section of the main classification is research character or type that includes 6 sub-categories which are validation research, evaluation research, solution proposals, conceptual proposals, opinion papers, and experience papers.

In light of the review process of study B, Engstrom and Runeson infer that system testing is the leading research focus with different contribution types such as solution proposals for designing test cases, the most popular research type

is solution proposals, and methods are frequently seen in the SPL testing field among other contribution types. In addition, they state that most of the works proposed in this field are presented in workshops and conferences, in which case studies and experiment-related papers are rare. Engstrom and Runeson correlate this scarcity with the fact that the cost of SPL testing assessments in general is high. Another important point they uncover is that a matter of testability is still not mature to be investigated. Last but not least, there is an urgency of innovative approaches to design test cases and make selections of them in SPL testing context.

Finally, Engstrom and Runeson suggest that, by taking the needs and challenges of SPL testing area into account, number of empirical works conducted in the field should be increased to provide logical reasons to the industry for encouraging them to apply newly proposed testing techniques.

Study C - Machado et al.

The purpose of conducting this systematic literature review, as Machado et al. indicate, is to identify, evaluate, and explain existing research evidence in SPL testing area, for the aim of clear-cut classification. The authors researched mainly 2 aspects of SPL testing in their study: the development of representative sets of products and related methods, and the available effort spent on testing application engineering products by making use of commonality and variability of the product line.

Related to first aspect, Machado et al. state that there are some proposed methods to solve the problem of selection of products from a large number of inputs, which are based on combinatorial techniques. Likewise, the primary studies mostly propose a solution that depends on either a process or an algorithm. Tool support of the methods proposed for the first aspect is not sufficient since most of them provide only a description of implementation's algorithm.

Regarding second aspect, the authors divide the primary studies into 4 groups based on characteristics the techniques given in the primary studies should cover:

variability, test reuse, automation and SPL process. In terms of variability, they point out that the test suite should include at least a number of test cases required to test each feature (resulted from the variability of the product line). For the matter of test artifacts reuse, they state that most of the works make a contribution of providing methodologies to cope with reuse of test cases and test scenarios. The automation of generation of test artifacts such as test inputs and test cases is an underdeveloped process since a small number of works is proposed in the primary studies. Lastly, Machado et al. indicate that some of the testing approaches given in the primary studies fit only one SPL process (domain engineering or application engineering) and some of them fit both.

Machado et al. emphasize that the existing research is mostly conducted as academic studies. Likewise, none of them eliminates the lack of guidelines to advance test engineers' skills to cope with design of tests and selection of proper ones for different cases.

Study D - Neto et al.

In this work, Neto et al. present a systematic mapping study that has goals of reviewing state-of-the-art testing approaches, interpreting available evidence, and indicating missing points between necessary techniques and current practices, existing in the SPL testing literature.

The outcomes of the process of extracting data in study D is given in 9 subsections with sub-strategies as follows:

- Testing Strategy: testing product by product, incremental testing, opportunistic reuse of test assets, design test assets for reuse, division of responsibilities
- Static and Dynamic Analysis
- Testing Levels: unit testing (core asset development), integration testing (core asset development), system testing (product development), acceptance testing (product development)

- Regression Testing
- Non-functional Testing
- Commonality and Variability Testing
- Variant Binding Time
- Effort Reduction: reuse of test assets, test automation tools
- Test Measurement

With a clear picture of above classification, Neto et al. infer that it is more popular in SPL testing research field to propose new methods as solution proposals and examining their properties than benchmarking them in practice. They notice that the SPL testing studies are lack of validation and evaluation research as well. Likewise, they believe that the industry needs much more experience reports to be issued related to the methods proposed in the SPL testing literature. Lastly, they underline the importance of domain specific languages and their tool support in the product line environment.

3.4 Discussions

In this section, we report the findings in relation to our research questions.

- *RQ1: What SLR studies have been published in the area of SPL testing?*

As an answer to this research question, we organize our data by paper reference, e.g. study A, in Table 3.3 3.6. We found 4 secondary studies that are type of either a SLR or MS. As it can be seen in Figure 3.3, 2 studies (50%) are in the category of SLR (study A, C) and other 2 studies (50%) are in the category of MS (study B, D).

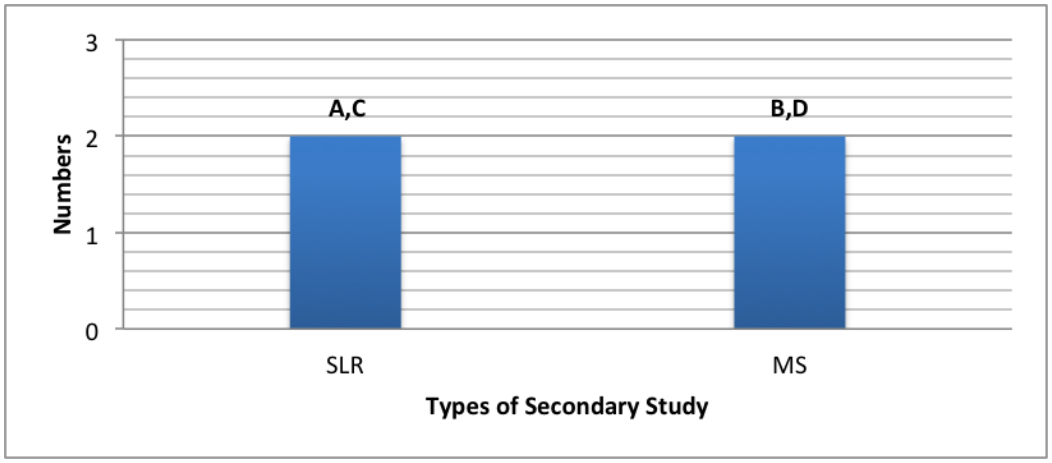

Figure 3.3: Type-wise distribution of the reviewed studies

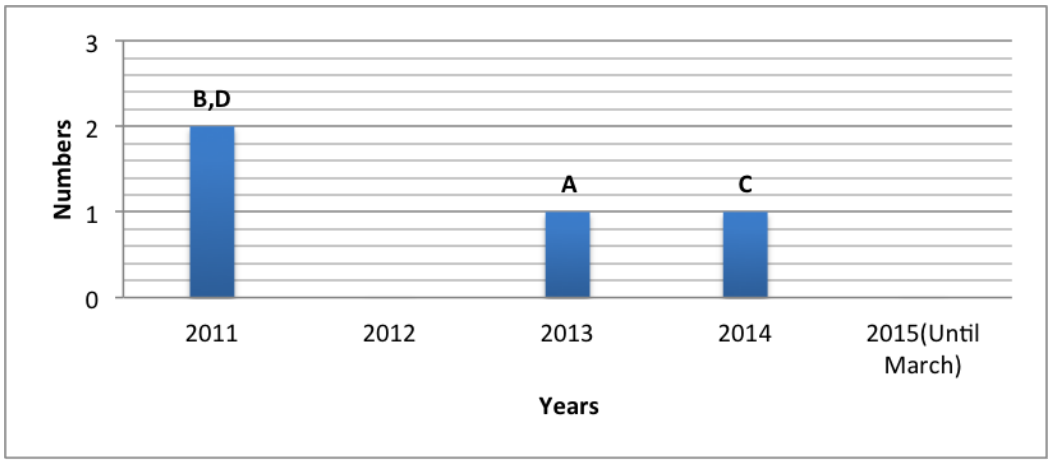

Figure 3.4: Year-wise distribution of the reviewed studies

The publication year of studies is varied from 2011 to 2014 as it is demonstrated in Figure 3.4. Two studies are published in 2011, one study is published in 2013, and the remaining one is published in 2014.

By looking the number of primary studies included in the secondary studies we reviewed, it can be said that the works proposed in SPL testing field are in abundance. The study A has the least number of primary studies that is 37 and the study B includes the most that is 64. The numbers are close for study C and study D, which are 49 and 45 respectively. Lastly, among the 4 studies, as a source, journal is dominant (75% "Information and Software Technology"); on the other hand, conference is secondary (25% "ICSOFT").

- *RQ2: What SPL testing research topics are addressed?*

In order to answer this research question we carefully analyzed all of the secondary studies. The general picture of the SPL testing research topics is composed of test levels, test organization and process including testing strategy, testing effort including test case derivation techniques, testability, test automation, measurement of test and test input, handling variability including traceability issues, variant binding time, and static and dynamic analysis. In addition, considering the research question, we present general overview of evaluations and suggestions, which the secondary studies provide, related to SPL testing field.

Test Levels

Testing of a software product is commonly performed at diverse stages or levels considering testing needs and purposes [39]. Three secondary studies (study A, B, D) we reviewed have a topic relating to test levels such as unit testing, integration testing, SPL architecture testing, system testing, acceptance testing, regression testing, non-functional testing, functional testing, and framework testing.

Unit testing aims to test each unit of software in isolation [39]. There are different proposed approaches for unit testing given in the studies. McGregor in [A1] specifies test cases and test plans based on units with corresponding reports that are elements of documentation that is part of a core asset. Likewise, in the same work, it is emphasized that when a core artifact is developed, related software units need to be tested. Ganesan et al. [A2] provide a work depends on an analysis of unit testing approach applied in a product line that is actually a framework composed of a set of variation points. Another approach is presented by Uzuncaova [A3], which generates test inputs for unit test of each product in SPL automatically using SAT-based analysis. The technique used in this work depends on mapping process of a formula that defines a feature as a transformation that specifies incremental clarification of test suites [A3]. Furthermore, some proposed studies include diverse approaches to generate test cases, considering requirements, including variability aspects. The main considerations in these works include covering likely scenarios. Reuys et al. [B1] propose a work,

called ScenTED, in which all likely scenarios are demonstrated using UML activity diagrams. In another study, parameterized use cases are used, which consider testing coverage criteria if it would be applied [B2]. Yet another method is proposed to create unit tests, which is based on an aspect-oriented approach [B3]. In addition, in study D it is underlined that, regarding testing levels, unit testing is performed in domain engineering. No additional information is given in the same study if unit testing is applicable to run in application engineering.

In order to test interactions between software components, the integration testing is a key. There is a prototype, named RITA Tool, given in [A4], which is used to perform integration testing in product lines and frameworks. Moreover, the tool is designed to deal with issues of commonality and variability in SPL. Reis et al. in [A5][B4] propose a technique of generation of test cases for integration testing in which all interactions of software components are covered. In addition to this, validation of the work is demonstrated in an experiment, as it is stated in the study A and study B. It is also important to test the integration between variations; however, testing of all possible combinations is not feasible. As it is indicated in the study A, in order to decrease number of combinations, combinatorial test designs can be effective. In [A1], two techniques are proposed for that purpose: one of them is based on combinatorial design and the other is for executing integration testing incrementally. The ScenTED method [B5] is also support integration testing, as it is stated in the study B. Furthermore, there is a method making use of model checking techniques to perform integration testing depends on test scenarios [B6]. As it is stated in the study B and study D, Li et al. demonstrate a way to create integration tests from unit tests, which is also shown in an industrial case study [B7][D1]. Likewise, in study D, it is pointed out that integration testing can be applied in both domain engineering and application engineering.

In regard to SPL architecture testing, only the study A mentions the current situation in SPL testing field. As a matter of fact, the information the study explains is related to improving testability by designing the SPL architecture considering all testing phases in the architecture design step [A6].

The system testing is interested in the characteristic of a system completely [39]. The studies we reviewed examine topic of system testing in different ways. The study A classifies its primary studies related to system testing as embedded system testing, for example. In addition, in the same study, it is noted that the works related to embedded system testing are bounded to a special domain and evaluated in a simulation or an industrial settings. Applying feature-oriented reuse method (FORM) and using an artificial environment, Kim et al. [A7] provide a tool that assists the development of SPL that is based on embedded systems of control. In another study [A9], a framework is developed to assess test cases regarding formal specifications and the technique is demonstrated with a product line related to remote control units. Kishi and Noda [A8] handle the verification of embedded systems design in SPL by applying model checking and corresponding tool is given as well. Pesonen et al. [A10], for smoke testing the devices, have a technique that makes use of aspects in embedded systems. They also provide the results of experiment with the Symbian OS. In [A11], an analysis is presented that is about the reasons of issues reported in Philips TV SPL testing phase. System testing is applicable to carry out in application engineering and can be performed on the software architecture by applying a static analysis approach, as it is highlighted in the study D. Likewise, the domain engineering could be the stage where system testing is performed [D2][D3][D4]. The main reason of doing so is to develop abstract test artifacts related to system testing in domain engineering and reusing them to derive products in application engineering, as emphasized in the study D. Engstrom and Runeson, in the study B, underline that the main aim of works related to system testing is automatically producing test cases from requirements that may be based on frequently use cases [B8], formal specifications [B9], or expressed in natural language [B10]. Hartman et al. provide a technique that depends on available UML-based tools [B11]. PLUTO, by Bertolino and Gnesi [B12][B13], is introduced to optimize tests based on use cases in SPL. Another proposed work derives test cases from use cases for domain engineering, while it retains the test cases' variability [B14]. Nebut et al. present an algorithm to produce test cases, which are product-specific, from requirements of the product line in an automatic way, which is demonstrated in UML [B15]. In [B16], Reuys et al. present the ScenTED method for creating

test cases from models based on UML. Another work is proposed by Olimpiew and Gomaa [B17][B18] in which UML notations are used such as tagged values, diagrams, and stereotypes. UML testing profile is included in another work as well [B19]. Kang et al. propose a work based on a variability model and UML use cases [B20]. Yet in another work, Weissleder et al. [B21] generate test suites using OCL expressions and reuse state machines. There are also two presented techniques that are composed of models to generate test cases, based on formal specifications of process algebra [B9][B22]. In a work [B23], using Alloy, an incrementally test generation technique is proposed.

In the study D, Neto et al. underline that acceptance testing can be carried out in application engineering. In the work [A12], an approach is proposed, which mainly uses executable acceptance tests as an explicit traceability link between code assets and feature models, for providing traceability links between the code assets and feature models. In another work [A13], Camara and Kobsa define a technique, which depends on SPL and aspect-oriented software development concepts, related to the generation of controlled online tests to analyze effects of webpage variants on users. Lastly, there is a proposed work, which is experimented in an industrial SPL, includes a decision model for acceptance testing [B24].

Regression testing is briefly performed to ensure that the software still passes the tests after any changes applied to it such as an update, which previously did [39]. In the study D, Neto et al. state that although the techniques for regression testing have been popular in terms of a research topic, there is not any evidence on these techniques applied to SPL. Likewise, they add that some existing studies give brief information about the importance of regression testing in a way that ignores the issues relevant to SPL aspects. In [D5], it is reported that any time a core artifact is modified, it has to re-tested using a set of regression testing. Another study [D6] emphasizes that if there is a correction or evolution related to software asset under the test, all test cases should be repeated during the process of regression testing. It is indicated by Kolb [D7] that one of the important issues of SPL testing field is related to regression testing. Lastly, Jin-hua et al. advocate that, regarding application architecture, whenever a component is

modified, regression testing becomes significant to perform [D8].

Neto et al. point out, in the study D, that non-functional testing is equally significant as functional testing and prove that some of the primary studies they analyzed introduce techniques related to generation or execution of non-functional tests. In [D9], the authors propose a method to assist the generation of reusable performance test scenarios that are later used in the application engineering phase. In [D10], the significance of non-functional concerns such as reliability and performance is underlined. Ganesan et al. express an approach to build a framework to test the response time and load of a SPL [D11]. Lastly, it is possible to apply non-functional testing methods, which are generally performed in a single-system testing, to the context of SPL testing, as it is stated in the study D. However, the authors add that there is not any experience report to support that idea.

Functional testing topic is investigated explicitly in the study A, which, according to the authors, is the most mentioned topic in the primary studies they analyzed. However, a majority of the information they provide belongs to topic of test case derivation in SPL. In regard to functional testing automation, the study [A14] describes a case study in which provided test strategies related to SPL testing automation are applicable to use in SPL in general. Furthermore, in the study A, Lamancha et al. share information about a primary study [A15], considering testing process of legacy systems, in which a set of generic test cases are created by reusing existing test cases. Lastly, another work [A16] is prepared in an experimental environment to examine the effectiveness of functional testing by finding diverse fault types in the reusable assets related to the commonality and variability of a SPL. According to the results of the experiment, it can be said that in terms of detecting variant-specific faults, the standard techniques are not efficient [A16].

Framework testing is included in the study B by referencing one primary study [B33] in which a model related to framework testing is presented. The model aims to application engineering testing phase and pinpoints not tested framework use cases that will later be covered using selected product test cases.

Test Organization and Process

This section includes studies mainly focus on mapping testing activities into SPL development phases properly, suggestions to organize testing processes or test planning steps, and testing strategies in SPL context. All of the four studies we reviewed have a section relating to this topic in general.

McGregor, in his work [B25][A1], underline that testing process in SPL context consists of testing core assets produced in the domain engineering, testing product-specific artifacts, and the interactions occurred between them. He also discusses a proper structure for test artifacts and necessary documentation related to that structure [B25][A1]. Pohl and Metzger [A17] list six fundamental steps in regard to SPL system testing and underline that they are necessary to follow when designing an approach in SPL testing context. The significance and complexity of a SPL is addressed in [B26][B27] by pointing the lack of guidelines and need for systematic testing techniques. In [B31], a conceptual work based on risks is proposed, which takes the attention to issues of test planning. Another work, depending on an abstract assessment framework, points a deficiency of test organization in SPL by stating that the perspective of customers is ignored in the context of SPL testing [B35]. In order to assist testing process, in terms of selecting defined features that will later be tested in diverse products, [B32] provides a story driven method. In the study C, Machado et al. discuss the SPL process existing testing techniques fit. They indicate that early testing in domain engineering is crucial since an unnoticed bug can be problematic in the further steps such as actual product development in application engineering. Moreover, they state that a complete test is infeasible in domain engineering due to large number of instances, testing of application-specific assets need to be performed in application engineering. Although they do not give the name or reference of the related works, they get 8 approaches applicable to perform both in domain engineering and application engineering, 10 methods applicable for domain engineering, and 9 methods applicable for application engineering. In the study D, Neto et al. suggest that the testing activities should be scattered over domain engineering and application engineering. Furthermore, they add that there are

some studies [D8][D12][D13][D14] in which the V-model is adopted as an applicable software development method for SPL and believe that this approach could be useful in terms of getting improved testing quality. In [B28], Tevalinna et al. highlight two issues related to dividing testing activities into different phases of SPL with respect to the V-model: firstly, thorough integration and system testing in domain engineering is not applicable, secondly, the domain testing might not be reliable in the application testing phase. In the context of SPL testing, a recent test model is also defined in [B29] named W-model. In another work [B30], testing of a product line that is based on agile methodology is discussed and also challenges of performing test driven development in SPL is pointed out. Tevanlinna et al. [D15][B28] present a set of testing strategies that can be followed in the SPL testing context. Likewise, in the study D, Neto et al. provide a comprehensive list of testing strategies including the ones given in [D15]. The first type they list is "testing product by product" in which there is not any testing process performed in the domain engineering; hence, it lacks reuse opportunities. In [D8], a similar approach is proposed, called "pure application strategy", in which testing is performed solely in application engineering. The second type is "incremental testing of product lines" in which testing of the first product is carried out and for testing of the other products, regression testing methods applied. The third type is called "opportunistic reuse of test assets" in which reuse benefits are considered only for application test assets and, as the name indicates, the strategy is not systematic. The next type is named as "design test assets for reuse" that is completely matched with the goal of SPL testing. Domain test artifacts are developed at the beginning to test common parts and to be ready for further testing variable parts. Those test assets are reused and modified later in the application engineering phase to test actual applications. The last type is called "division of responsibilities" that means selection of test levels that are further performed in domain and application engineering phases should be done considering each SPL process's goal that is either developing for reuse or developing with reuse.

Testing Effort

The topic of testing effort is given in the study D as a combination of two aspects of test effort reduction approaches that are "reuse of test assets" and "test automation tools". In the same study, it is underlined that testing process of common parts and actual products individually makes the SPL testing as a whole costly. Considering test effort reduction strategies related to "reuse of test assets" types, it can be said that these strategies mainly depend on reuse of test cases, test scenarios and test results. As it is underlined in [D6][D16], in order to benefit from the reuse of common parts in the domain engineering, there should be an artifact repository so that the cost of generation of test artifacts initially would be eliminated. Yet in another strategy, as it is highlighted in the study D, it is advised that in domain engineering, the test artifacts should be generated in a wide range as possible and the potential variability should be assumed by considering to develop abstract test cases and document templates, as well. In [D17], a technique is defined to examine commonality-related problems to eliminate iterative executions, which mainly depends on monitoring the status of each component's interfaces through the test execution. Neto et al., in the study D, also emphasize that many studies [D18][D19][D20][D3][D21] focus on the reuse of test assets in a systematic way to decrease test effort. Likewise, they note that there is not a common solution for coping with systematic reuse-related concerns in the context of SPL testing. In the study C, Machado et al. state that the characteristic they called "asset reuse" is popular among the primary studies they analyzed although the authors do not indicate whether the primary studies touch the topic of testing effort. They also add that the characteristic is composed of reuse of test data, test cases, test scenarios and test results considered either in domain or application engineering. Furthermore, they mention a testing technique, which is examined in [C2][C3], based on delta-oriented approach that points out the modifications to be applied to a core module composed of common parts of the SPL to create actual products [C1]. In this context, as it is underlined in the study C, test assets of a new product are created considering the deltas between this and the instances of products pre-created; hence, the testing effort is reduced by focusing on the deltas, i.e. reusing test assets. As an interesting inference provided in the study C, Machado et al. state that while the approaches given in the primary studies they reviewed are mostly dealt with reducing the

testing effort, they ignore the topic of getting higher level of fault detection rates. The strategies related to "test automation tools" are essentially proposed for single system testing context and can be mapped to SPL testing field in terms of functional testing, as it is indicated in the study D. Likewise, it is also a hot topic in the study C in which Machado et al. indicate that a minority of the primary studies they analyzed gives information about how to automate the development of test assets. They also add that the primary studies (19% of all primary studies they reviewed) mention their tools but there is not any concrete tool provided.

In the study A, Lamancha et al. approach to the topic of testing effort as predictors of testing effort. In [A18], Ajila and Dumitrescu prepare a work related to a product line, and infer that in terms of an effective predictor of testing effort, the code size is not efficient. In the study B, Engstrom and Runeson have information related to factors that affect testing effort in SPL context. They state that there is a study provides cost models in relation to the factors the study identifies and have effects on SPL testing effort [B34]. In another work [B36], a fault model, which is related to the defects possible to occur in a product instance, is introduced for reducing testing effort.

In this section, we also present existing test case derivation techniques that are useful to reduce testing effort in the SPL testing context. There are different proposed techniques in the literature to drive test cases such as [A19][A1][A20][A21][A22][A23]. In the study [A19], the proposed method, called "PLUTO Product Line Use Case Test Optimization", represents variability with UML tags and covers use cases in the SPL. The way it derives test cases is based on Category Partition technique. An identical approach is given in another work [A20] in which the dependencies are expressed with associating contracts to the SPL use cases and UML tagged values, and the necessary pre/post conditions are given as UML notes. In addition to this, a tool is introduced in the work to automate test case generation. McGregor has a method in which use case scenarios are used to generate generic test cases [A1]. Afterward the product related test cases are created from the generic test cases and issues related to the variability are handled using the orthogonal arrays approach. A model-based approach to SPL testing is proposed to develop test specifications from use cases and feature

models [A21]. In another study [A23], the proposed method, which is performed in the industry as well, aims to create test cases from use cases with the help of sequence diagrams. In [A22], the proposed method provides reuse opportunity of test cases created in the domain engineering phase of a framework, which is crucial to cover framework use cases in the application engineering phase of the same framework. In the study [A20], it is also proposed to create an abstract view of test scenarios, called "behavioral test patterns", from sequence diagrams. Kang et al. [A24] generate test scenarios from modified sequence diagrams and a test framework based on Orthogonal Variability Model (OVM) [43]. The reason why they changed the notion of standard sequence diagram is to include variability. In another work [A25], an automated testing framework based on standard metamodels and model transformation languages is presented for SPL testing, which aims for creating test cases from UML sequence diagrams using model transformations. In the study [A26], in order to develop test cases, a workflow based on the system's model is used.

Testability

The subject of testability of product line is investigated by McGregor [B25] in his report that relates technical aspects of SPL testing concepts. In [B37], in order to guarantee a SPL's testability, design policies are needed since testing cannot catch a set of defects as Trew underlines. In another work [B38], a technique is presented to advance and assess testability relating to SPL architecture.

Test Automation

A technique of designing test automation software is presented and assessed in the work [B39]. The software takes variability of product asset and test asset into account. The study [B40] presents an approach, which is domain-based, to automate testing of product line. In another work [B41], the traceability is found between tests of variability and its implementation by investigating the way to combine unit testing and aspect-oriented development. Yet in another study [B42], a technique is proposed to combine a product line of control systems and test automation.

Measurement of Test and Test Input

One of the most important practices in SPL testing belongs to the concept of measurement of test and test input. Test measurement is applied to balance required effort of carrying out a method. In this context, there are different test coverage criteria proposed in the testing literature. According to [D22][B45], making use of covering arrays strategy is one of them. Another strategy to evaluate adequacy of testing in framework-based product lines is presented in [D23][B46], which consists "hook" and "template" coverage. In [B25], it is discussed to have a structural coverage by combining each product variant's test executions in a SPL. By considering the aim of a coverage criterion, in [B43] process of SPL testing is improved by diminishing set of adjustments that are necessary for verifying the platform's variation. Many algorithms are developed by Gustafsson [B44], which mainly aim to cover completely a product line's features in minimum one product instance.

Test input measurement becomes crucial when the number of elements in a set of test inputs is huge, such as in the case of SPL testing where the number of configurations to be tested considering all variation points, variants, etc. is really large. Therefore, there are various strategies including different types of coverage criteria proposed to reduce number of test inputs to a reasonable limit. In the study C, Machado et al. underline that the primary studies they analyzed propose techniques based on "combinatorial criteria on feature models", "coverage criteria on variable test models", or "coverage criteria on feature interactions". Likewise, they add that approaches based on t-wise feature coverage combined with SAT solvers are prevalent in the literature. Application of pairwise testing, which is derived from t-wise coverage, exists in 50% of the primary studies Machado et al. reviewed, as they state. Furthermore, they find it important to note that the tool support in this context is provided mostly as descriptive information.

In the study [A27], a strategy, which relies on pairwise as coverage criteria, is proposed for testing assets that come from feature models of a SPL. In another work [A28], a graph-based testing method is presented for the process of selection of assets and features, and for testing them. In order to select test products, a

combination of combinatorial testing, forward checking, and graph transformation is used in [A29]. Depending on data-flow coverage criterion and definition/use technique, a strategy is proposed in [A30] for selecting covered data dependencies that are not needed for testing again. Lastly, in another work [A31], t-wise feature coverage integrated with SAT solvers is used.

Handling Variability

In this section, we provide approaches defined for handling variability in SPL context and discuss issues related to traceability that is important for SPL testing field.

The variability can cause a large number of instances to test in the SPL since each variation point offers a new combination of features to create an asset. As a result of this situation, a trade-off is appeared between a wide-range of variability, i.e. having a diverse set of products developed from the SPL, and testability of the assets. Therefore, techniques to handle variability become a must. The study [D22] presents cumulative variability coverage, which progressively collects data of coverage in each of the development stage, to be later used in testing phases of other assets that is intended for testing. Yet in another work [D24], in regard to software architecture, the idea is to isolate variability from commonality and represent the variability as a subcomponent, rather than having a component with huge amount of variability that will later require a high level of effort to test.

Regarding artifact traceability from requirements stage to the code generation stage in the SPL testing context, the study [D15] underlines the significance of this process. It is highlighted in [D25] that if the implementation structure of a SPL asset's variation is matched the design of each SPL test asset, it is possible to obtain the traceability. In the study C, Machado et al. emphasize that algorithms, called "traceability-mining" [C4], are useful to fix and examine the tracing data, but none of the primary studies they reviewed considers this case as they state. Furthermore, they believe that there should be traceability between feature and test code since it is significant to have information about functionalities of each

SPL product that is eventually composed of its features and source code. The two studies [C7][C8] they reviewed investigate this case. In addition, Machado et al. highlight the fact that the traceability links between test code assets and models related to test assets might be outdated as the overall system is modified or evolved. Therefore, the consistency between assets should be under control. The work [C5] presents a method to keep traceability links between a feature model and test code assets up-to-date as the variability represented in the feature model evolves. In [C6], in order to link a test case to a feature, the idea is combining the test case with an attribute so that the traceability between them is maintained.

As it is indicated in the study D, variability can be handled with different methods such as requirement-based techniques in which variability is referred in UML use cases that are later used to generate test cases, model-based techniques in which test models are included variability, and approaches that encapsulates variability in feature models and activity diagrams.

Variant Binding Time

In the study D, Neto et al. state that distinct binding times such as compile time, execution time, etc. are required for binding of distinct variants. In addition, the binding times depend on distinct instruments such as inheritance, overloading, etc.; hence, different forms of faults might occur depending on the instrument and as a result of this, different test techniques might be needed. This issue is highlighted in the work [D12][B47] in which a model, called "Variability and Testability Interaction Model", is presented. The interaction between testability and variant binding is examined using that model.

Static and Dynamic Analysis

The process of quality management in SPL context relies on complete usage of static and dynamic analysis approaches in order to be thorough and effective. In the study D, Neto et al. underline that most of the primary studies they reviewed indicate that in the context of SPL testing, static analysis methods such as formal verification techniques, inspections and walkthroughs should be

performed before dynamic analysis techniques. In [D5], an approach is provided for "Guided Inspection", which has a goal of regulating review process of non-software artifacts. In [D26], a model checker is presented which targets design verification rather than code verification. In order to organize dynamic analysis in a SPL, the studies [D12] and [D24] suggest adopting V-model stages. Lastly, in the study D, Neto et al. point out that strategies related to managerial and technical aspects have a key role in adjusting the effort spent between static and dynamic analysis activities. They add that, from a technical perspective, the focus is determined by the development type such as test-first or model-based; the former points out dynamic analysis, while the latter points out static analysis. From a managerial perspective, they state that strategies related to managerial aspects, such as time-to-market and product quality, affect the decision to which analysis is performed.

Evaluations and Suggestions Related to SPL Testing

According to the study A, popular topics in the field of SPL testing are functional testing and variability testing (i.e. handling variability) with UML models and use cases; on the other hand, the SPL aspects such as test planning, effort, management and oracle definition are not mature and should be taken into consideration by the researchers for further improvements. For test automation processes, there are different approaches proposed, however, they are mostly prototypes. Considering general testing process in the SPL, it is underlined in the study A that the existing primary studies provide guidelines instead of defining a solid process. In addition, model-based testing and approaches related to generation of test input are fundamentals and promising for SPL testing field.

In the study B, the authors indicate that while researches are mostly focus on system testing and handling variability, the topic of testability is barely taken their attention. Moreover, issues related to test management are mentioned in the remarkable proportion of the primary studies they analyzed although the topic of adjusting the required effort between testing in domain and application engineering is not popular. Last but not least, there is an urgency of innovative approaches to design test cases and make selections of them in the SPL testing

context and new methodological approaches with effective tools.

Machado et al., in the study C, highlight that despite presented tool support for SPL testing, the tools are not mostly available in the field. Furthermore, they note that there is not much ongoing research on model verification task. They also indicate an interesting deficiency exists in the SPL testing field, which is the lack of guideline/methodology that could make test engineers an expert of test design and test selection considering the present testing conditions.

The trend in the research of SPL testing, according to the study D, is in the direction of proposing new methods rather than evaluating their practical use and expressing their experiences of using these methods, providing solution proposals, and frequently dealing with topics of testing strategies and testing levels. They also provide a list of topics in which promising solutions are required such as regression testing, variant binding time, measures, non-functional testing, and static-dynamic analysis.

All studies indicate that the SPL testing area is lack of empirical studies and results. Likewise, comparative studies are needed, in which the proposed techniques are compared and benchmarked to point out which method among/against others is most suitable for given conditions.

In Figure 3.5, the secondary studies we reviewed in this work are demonstrated according to SPL testing topics they analyzed. It can be stated that topics of test organization and process, testing effort, and measurement of test and test input are popular among the secondary studies. Each of testability, test automation, and static and dynamic analysis aspects of SPL testing is included only in one study.

- *RQ3: What research questions are being investigated?*

The bulleted list given at the end of Section 3.3.1 presents the research questions from each secondary study. By analyzing them, it can be underlined that

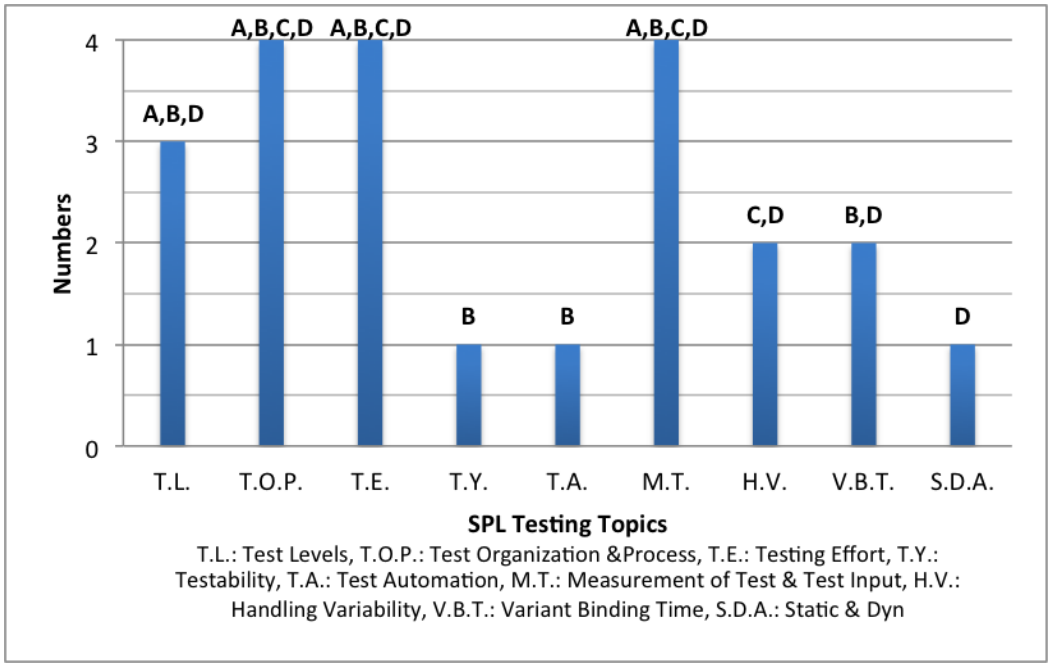

Figure 3.5: Topic-wise distribution of the reviewed studies

each study at least aims to investigate existing techniques, initiatives, or strategies related to challenges and topics of SPL testing field. In addition to this, some studies are curious about the venues in which studies related to SPL testing area are published, studies' research types, quality of existing research, and trends for future research. Lastly, although the number of reviewed SLR in this tertiary study is limited due to some reasons such as immaturity of the research field in terms of available systematic reviews and our inclusion/exclusion criteria applied to the studies resulted from the search process, it is possible to state that the secondary studies we analyzed cover most aspects of SPL testing field in terms of both technical and research-related parts.

3.4.1 Threats to Validity

The fundamental aim of conducting a systematic review is to identify the current state of a targeted field, which is SPL testing in our case. The types of validity possibly considered in this tertiary study are construct validity, internal validity, external validity and conclusion validity [44].

The construct validity mainly focuses on the connection between the theory behind the study and the observations [44]. It is related to the study's design, the way it represents what the researches have in mind, and the phenomenon investigated according to the research questions. In order to eliminate possible threats to construct validity, in the search string we used keywords of "testing", "software product line" and "systematic literature review" as main constructs and detailed them with their possible synonyms. Likewise, for the construction of first and second construct, we took advantage of performing pilot searches. We followed the guidelines [22][35] to develop different and valid versions of the third construct. Furthermore, in order to develop research questions, inclusion/exclusion criteria and quality assessment criteria, we obeyed related procedures of the guidelines [22][30][35]. Considering construct validity, it is also important to find relevant studies in the topic under investigation as much as possible. Therefore, we searched automatically different electronic databases and publication venues manually.

The internal validity is about being able to ensure that the results of the conducted work are caused by only the one variable under study, not by any other variables [44]. The subjective decisions can be considered as threats to internal validity during the stages of secondary study selection and data extraction. Likewise, possible bias in evaluation of studies can be a threat to internal validity, as well. In order to avoid potential threats, we decide to follow the procedures defined for study selection and data extraction processes provided in the well-known guidelines [22][35]. In addition to following guidelines, we carried out pilot data extractions to create a base for data extraction process. In regard to threats to validity of assessment of studies, we used a pre-defined protocol to evaluate the studies iteratively.

It can be said that the generalization of results of the study under investigation, considering the limits in which they are valid, is the concern of external validity. We tried to find systematic reviews that identify aspects of SPL testing field. Generalization of our findings, in terms of the secondary studies resulted from the search process, might be hard due to a possible lack of relevant search string keyword related to "SPL", "testing", "SLR", etc. used in the search process,

which results in a limited number of systematic reviews to generalize in the end.

The conclusion validity means the chance of obtaining same results when replicating the same experiment. In order to achieve this aim, we followed the procedures of conducting a systematic review defined in the guidelines [22][35]. However, it is possible that other researchers could not obtain the same results at the end of the process due to the subjectivity exists in the assessment of the secondary studies.

3.5 Conclusion

Through a tertiary study, this work identified which aspects of SPL testing have been researched and investigated through systematic reviews including types of SLR and MS. In the end, after carefully analysis of the reviews, this section presents the findings.

The literature search provided us different studies including works related to SPL testing context and irrelevant papers mostly. After the inclusion and exclusion criteria applied, the number of studies decreased sharply to 4 to analyze. According to the data extracted from the secondary studies, distribution of study type was equal: 2 studies based on systematic literature review and 2 studies based on mapping study.

The SPL testing research topics addressed in the studies are test levels, test organization and process including testing strategy, testing effort including test case derivation techniques, testability, test automation, measurement of test and test input, handling variability including traceability issues, variant binding time, and static and dynamic analysis. All of the secondary studies we analyzed have information related to the topic of test organization and process, testing effort, and measurement of test and test input. Only one study investigated the topics of testability and test automation in detail, and the subject of static and dynamic analysis was exclusive to another study. We also presented the overall assessment

and suggestions related to the field.

Regarding our third research question, it can be noted that each study at least aims to investigate existing techniques, initiatives, or strategies related to challenges and topics of SPL testing field. In addition to this, some studies are curious about the venues in which studies related to SPL testing area are published, studies' research types, quality of existing research, and trends for future research.

Finally, with the help of all information provided in this tertiary study, it is our hope that existing knowledge about the SPL testing field related to its current state, challenges and needs will get a boost and researchers, also practitioners, will be encouraged for further research on this field with empirical works.

Chapter 4

Reflexion Modeling for Software Product Line Engineering

The reflexion modeling technique is applicable to use in the software product line engineering (SPLE) methodology that offers two different components in the architectural level, which are reference architecture, i.e. product line architecture, and application architecture. The former is created in the domain engineering phase and provides commonality and variability of the product line; the latter is created in the application engineering phase and derived from the former. In addition, the commonality entirely exists in the application architecture but the variability is decided according to the application requirements. The point where the reflexion modeling technique is introduced for SPLE is testing of conformance of the application architecture to the reference architecture.

In this chapter, we present enhancements we made to the general reflexion modeling idea to use it in the SPLE concept and general process of the approach we developed for consistency checking between product line architectures. The technical explanation of our approach such as tools used in the approach, developed code segments, structure of generated XML file related to the reflexion model, etc. will be given in Chapter 6.

4.1 Enhancements of the Reflexion Model

In the context of SPLE, the reflexion modeling technique needs an enhancement that takes an important aspect of SPLE, called deltas, into account. A delta is an entity that is not part of core assets in the domain of product line but is required by a specific application derived from that product line [14]. In other words, a delta element is not included in asset pool of the product line where common and variant elements exist. The enhancement we made in the reflexion model is that we indicate parts that are given in the application architecture but are not given in the reference architecture as delta instead of divergence (see Table 4.1).

Another enhancement we considered is related to concepts of convergence and absence given in the reflexion model. Since the conformance checking between a reference architecture and an application architecture is done by considering common and variable parts they share (also including deltas the application architecture has), the result given in the reflexion model as convergence and absence parts should be divided into two parts as common and variant. Therefore, we extend context of convergence and absence as (see Table 4.1):

- Convergence: Common Convergence - Variant Convergence
- Absence: Common Absence - Variant Absence

The concept of Variant Absence, in our work, is not only used to define the elements that are given in the reference architecture but does not exist in the application architecture, but also used to define missing elements in the application architecture due to constraint violations that are defined in the reference architecture (see Section 4.2). The reason why we made this enhancement is that this type of missing elements in the application architecture causes a type of absence, which is actually the absence of correct and expected form of the application architecture that would conform to the reference architecture if there would not any constraint violations.

Table 4.1: Enhancements of the reflexion model

Consistency Rule	**Description**
Common Convergence	Convergence elements that are given as common elements in the reference architecture.
Variant Convergence	Convergence elements that are given as variant elements in the reference architecture.
Common Absence	Absence elements that are given as common elements in the reference architecture.
Variant Absence	Absence elements that are given as variant elements in the reference architecture.
Delta	Elements that are given in the application architecture but do not exist in the reference architecture.
Not Selected Variant	Variant elements that are not selected in the application architecture.

In order to give complete information related to the status of each element of the reference architecture, we also added an additional identifier for not selected variants available in the reference architecture to the reflexion model designed for SPLE. For instance, if there is an optional variant in the reference architecture, which is not selected in the application architecture, the generated reflexion model indicates that variant as a not selected variant (see Table 4.1).

It is noteworthy to mention how the three inputs of reflexion modeling technique, which are high level model (HLM), source model (SM) and mapping rules, are represented in our approach. The HLM and SM are taken explicitly as following:

- HLM: The reference architecture
- SM: The application architecture

The mapping rules, however, are defined implicitly in a way that they are given in reference architecture metamodel definition with respect to the architectural viewpoint. The main motivation of doing so is related to nature of the reference architecture that defines how structure of derived application architectures can

be, i.e. which element of the application architecture can be mapped to which element of the reference architecture.

4.2 General Process of the Approach

The approach has six main steps, which is demonstrated as an activity diagram in Figure 4.1. The first four steps are done in parallel. It begins with providing reference architecture metamodel definition and application architecture metamodel definition. Language used to provide the metamodel definition is Emfatic [45] and all generated metamodels are based on EMF [46]. Common part of the metamodel definition considering all architectural viewpoints is the part where variability of the software product line (SPL) is introduced with Orthogonal Variability Model (OVM) [3] that is explained below in the minor section called "OVM and Its Metamodel". In the approach, the generated metamodels are modified if it is necessary; otherwise, reference architecture model and application architecture model that must conform their metamodels respectively are created. The models are created using Human Usable Textual Notation (HUTN) [47], which are single files with extension ".model". If the modification is required for generated models, it is done before next step of the approach. Whenever a model is generated, validation between it and its metamodel must be done to check their conformance. The language we used to validate conformance of metamodel-model pairs is Epsilon Validation Language (EVL) [48]. If the validation is not successful, the problematic parts of the model/metamodel must be modified. If it is successful, next step of the approach is transformation of models into an XML [49] file, which is a model-to-text (M2T) transformation done by Epsilon Generation Language (EGL) [50]. The reason why we transform models into an XML file is that it is easy to handle XML files in terms of both reading and writing programmatically and can be shared if their schema is provided. The generated XML based models are then read by our tool that is developed using programming language Java and conformance analysis of the reference architecture and the application architecture is done according to pre-defined rules that are specific to each architectural viewpoint and will be explained in detail in the

next sub-section. After the conformance analysis step, our tool generates two files: first one is an image file with extension ".png" that represents the resulted reflexion model graphically and second one is a detailed XML based file that represents the resulted reflexion model textually, which also includes statistical data related to number of convergence, absence, delta elements and not selected variants available in the reference architecture with their percentage.

OVM and Its Metamodel

The OVM is introduced by Pohl et al. [3] to define the variability of a SPL. It is a separate model that includes the variability information. The basic elements of OVM are variation point and variant. A variation point is a point at which the variation occurs, which also must be associated with at least one variable item, i.e. variant.

There are different proposed metamodels for OVM in the literature, two of them [3] and [51] provide comprehensive enough and almost identical metamodels, which assist us to create a new one according to our approach's needs. We define the OVM metamodel once and then we combine it with the metamodel of each architectural viewpoint we used in this thesis and the result obtained from the combination is the architectural viewpoint metamodel that can be used in a SPL. By doing so, it becomes easy to define and modify reference architecture model in which the variability of the SPL is defined. In addition, the architecture conformance analysis process of the approach is relieved since handling two input files programmatically is easier than handling three input files (in this case, there would be one extra input file for variability metamodel defined with OVM to the process).

In Figure 4.2, the OVM metamodel combined with the metamodel of decomposition viewpoint for the reference architecture is shown, for instance. The metaclasses called "Model, Element, Module, Subsystem, Property" belong to the decomposition viewpoint metamodel and will be explained in the next subsection. The metaclasses other than the ones belong to the decomposition viewpoint metamodel form our OVM metamodel. The integration point at which the

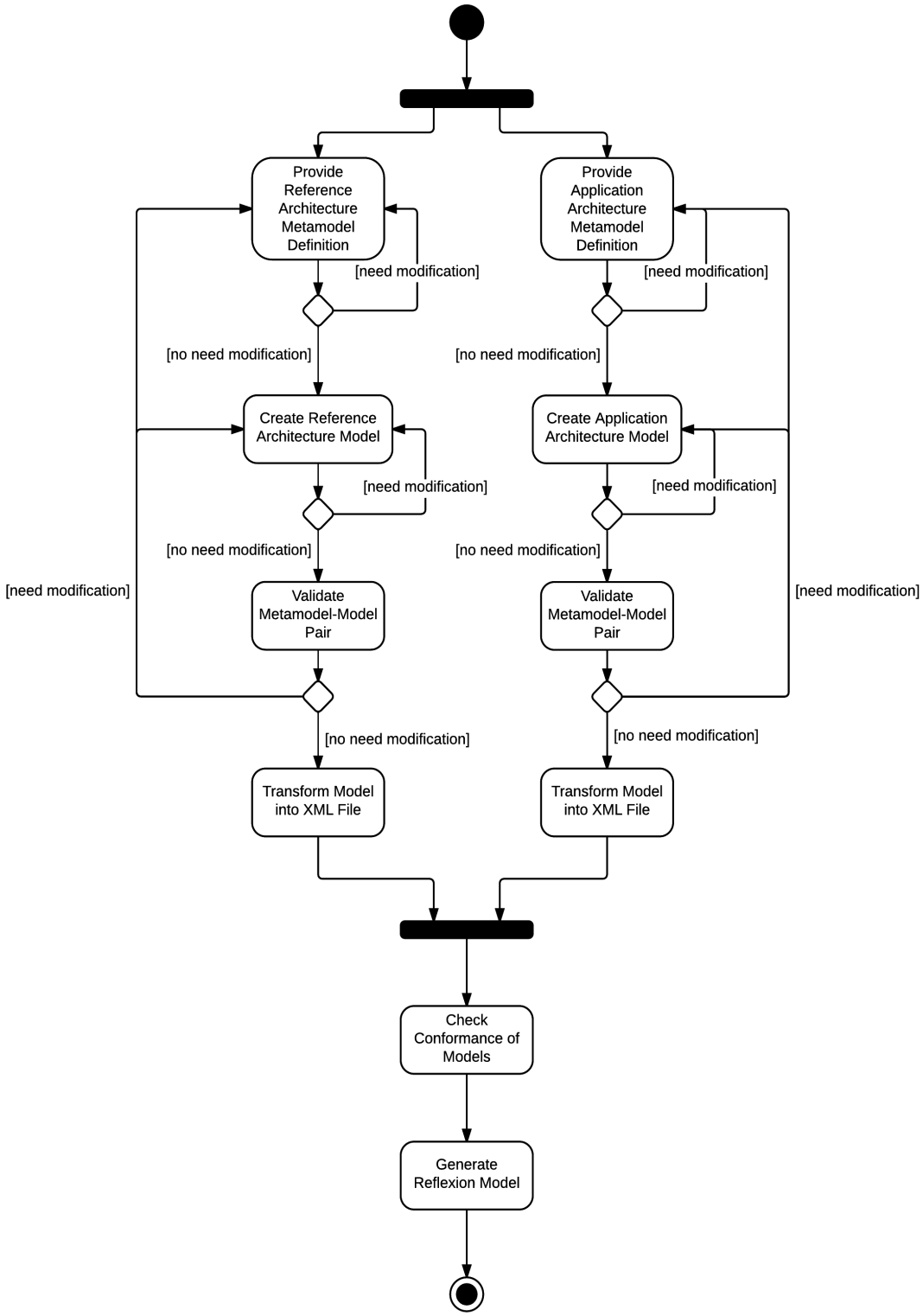

Figure 4.1: Overview of our approach

OVM metamodel and any viewpoint metamodel are joined is the metaclass called "Model" that is common in all viewpoint metamodels. The rationale behind this integration point is that we design each reference architecture metamodel, free from the viewpoint type, in a way that every model conforms the metamodel must have a commonality and might have variability since in a SPL there is always at least one common asset; however, the variability is optional, i.e. it is possible not to have any variable assets. Therefore, the multiplicity of the relationship "commonality" is one and the multiplicity of the relationship "variability" is zero to one, i.e. optional. All metaclasses of the OVM metamodel are explained as follows:

- Metaclass "CommonParts"

It represents all common elements given in the SPL and also has a direct relationship, called "commonelements", to element of the viewpoint with multiplicity one to many that indicates the viewpoint must have at least one common element.

- Metaclass "VariableParts"

It represents variable elements given in the SPL and is composed of constraints and variation points. The variability of the SPL can have zero or more constraints and zero or more variation points; the latter case can be seen when variation points are planned but not yet defined in the SPL.

- Metaclass "Constraints"

There are three essential variability constraint types in OVM [3]. The first group, called "Variation Point Constraint Variation Point (VP_C_VP)", is occurred between two variation points and has an association link called "uses" with metaclass "VariationPoint". In addition, it is either "Variation Point Requires Variation Point (VP_R_VP)" type or "Variation Point Excludes Variation

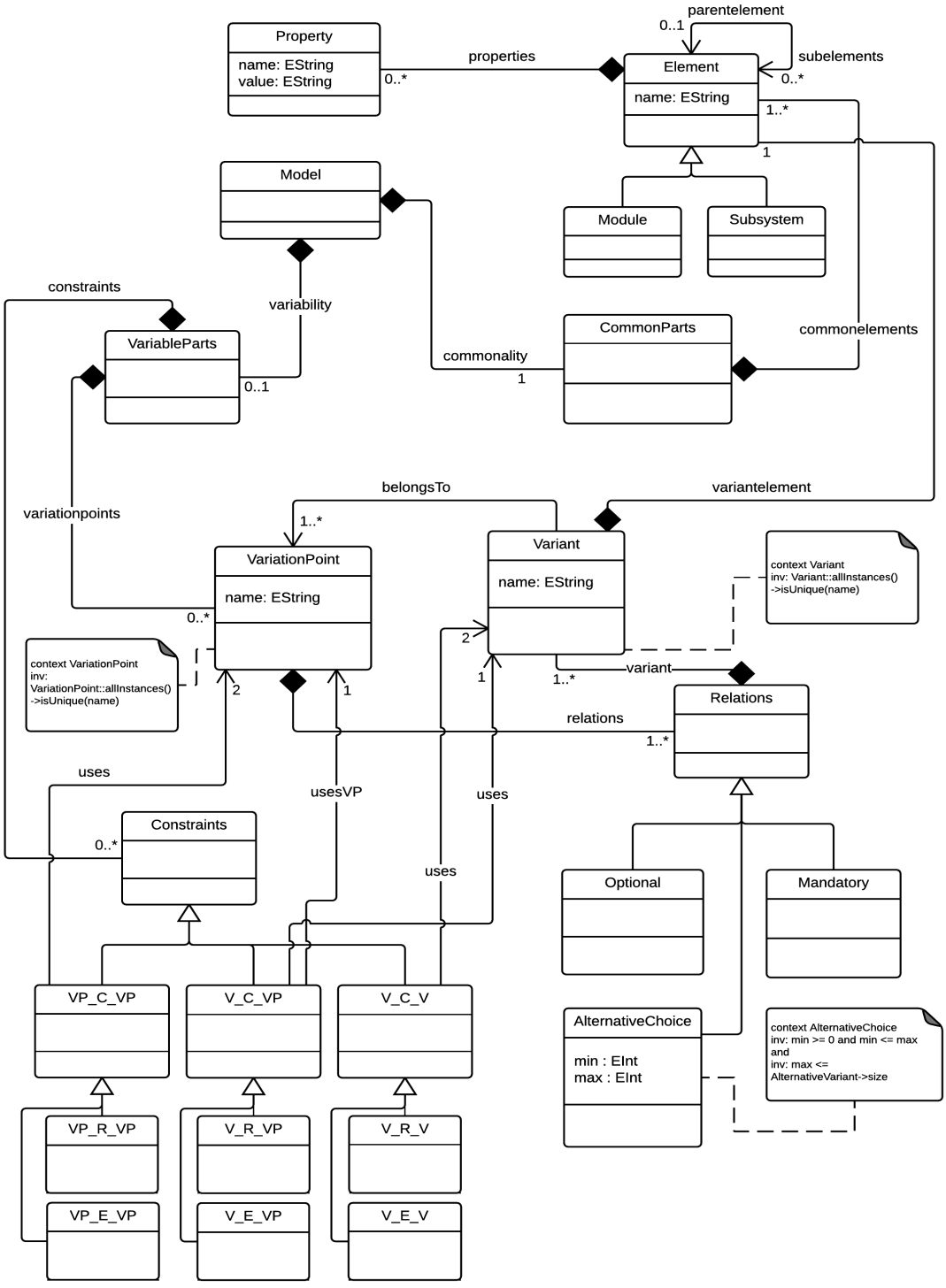

Figure 4.2: OVM metamodel combined with decomposition viewpoint metamodel for the reference architecture

Point (VP_E_VP)" type. The type of "VP_R_VP" means that in the SPL if the first variation point of the constraint is realized, i.e. at least one of its variants is realized in the application architecture, then the required second variation point of the constraint must be realized; otherwise this is a violation of that constraint. In case of "VP_E_VP" type, the second variation point of the constraint should not be realized in the application architecture if the first variation point of the constraint is realized; otherwise this is a violation of that constraint. The second group, called "Variant Constraint Variation Point (V_C_VP)", is occurred between one variant and one variation point, and has an association link called "usesVP" with metaclass "VariationPoint" and another association link called "uses" with metaclass "Variant". It is either "Variant Requires Variation Point (V_R_VP)" type or "Variant Excludes Variation Point (V_E_VP)" type. In the former, selection of the variant requires realization of the variation point given in the constraint. In the latter, selection of the variant excludes realization of the variation point given in the constraint. The last group, called "Variant Constraint Variant (V_C_V)", is occurred between two variants and has an association link called "uses" with metaclass "Variant". Same as other constraint types, this constraint is either "Variant Requires Variant (V_R_V)" type or "Variant Excludes Variant (V_E_V)" type. As it is stated in [24] that this constraint type is independent of the variation points the variants are associated with. The "V_R_V" type indicates that selection of the first variant of the constraint requires selection of the second variant of the constraint. The "V_E_V" type indicates that selection of the first variant of the constraint excludes selection of the second variant of the constraint. Lastly, any violations of these constraints are signaled by our tool.

- Metaclass "VariationPoint"

It represents all variation points given in the SPL. The metaclass has an attribute for setting a name for the variation point. Each variation point has at least one relation, which is shown with the composition relationship "relations" in the OVM metamodel.

- Metaclass "Relations"

It represents all relations in a variation point that is given in the SPL. The relations are three types, which are mandatory, optional and alternative. If a variation point has a mandatory relation and that variation point is realized in the application architecture, then the variant of the mandatory relation must be selected in the application architecture; otherwise this is a violation that will be signaled by our tool. Optional relations indicate that selection of the variant of the relation is optional, i.e. it is possible to not select an optional variant. The alternative relation is similar to optional relation with one exception that it has a constraint of minimum number of variants to be selected and that of maximum number of variants to be selected. If there are four variants in an alternative relation of a variation point in the SPL and that variation point is realized in the application architecture, and has the constraint of minimum number of variants to be selected as two and that of maximum number of variants to be selected as three, then at least two variants of the variation point and at most three variants of the variation point must be selected in the application architecture. These constraints are represented as attributes "min" and "max" in the metaclass "AlternativeChoice". In order to relate a relation with its variants, there is a relationship called "variant" between metaclasses "Relations" and "Variant" in the OVM metamodel.

- Metaclass "Variant"

It represents all variants listed in a relation and, indirectly, in a variation point given in the SPL. The metaclass has an attribute called "name" to set a name to the variant. In addition, the association link called "belongsTo" maps the variant to the variation point, which is associated with in the SPL. In addition, each variant is represented with exactly one element through relationship "variantelement" in the metamodel.

The UML notes in the OVM metamodel are just basic OCL [52] constraints that are implemented using EVL in our tool. The OCL constraint linked to the metaclass "VariationPoint" indicates that each variation point given in the SPL must have a unique name. Another OCL constraint linked to the metaclass

"Variant" indicates that each variant given in the SPL must have a unique name. The last OCL constraint linked to the metaclass "AlternativeChoice" underlines that value of the attribute "min" can be at least zero and at most equal to value of the attribute "max", and value of the attribute "max" can be at most equal to the number of variants given in the alternative relation. Any violations of these constraints are signaled by our tool.

In the process of architecture conformance analysis, for a given case, we assume that the OVM included in the reference architecture viewpoint of the SPL is designed properly such as there is no cycle in the constraints related to the variable part of the model. For example, let's assume we have a case in which two constraints are defined related to two different variation points, lets say VP1 and VP2, of the case's reference architecture, which indicate that VP1 requires VP2 and VP2 excludes VP1. In this case, this is a design mistake in the OVM and thus, it is a cycle (VP1→VP2→VP1). Our approach does not check this type of design mistakes and will lead to wrong results. Therefore, it is important to note that the OVM of the case must be designed in a way that it has not any erroneous points in terms of its design and structure so that our approach can work correctly and produce reliable outcome.

When we perform the architecture conformance analysis step of our approach, if there is a defined constraint in the variable part of the reference architecture and it is violated in the application architecture, we do followings:

- If the violated constraint is "VP_C_VP" type, then the required or excluded variation point's all variants are placed into the variant absence list.
- If the violated constraint is "V_C_VP" type, then the required or excluded variation point's all variants are placed into the variant absence list.
- If the violated constraint is "V_C_V" type, then the required or excluded variant is placed into the variant absence list.

In the same step, if there is a violation in the application architecture related to value of "min" and "max" attributes of the alternative relation (same constraint

violation as mentioned above related to OCL linked to metaclass "Alternative-Choice"), all variants listed in that alternative relation are placed into the variant absence list.

Chapter 5

The Approach and the Software Architecture Viewpoints

There are three distinct fundamental types of architectural style introduced by Clements et al. [4], which are Module styles, Component-and-connector styles and Allocation styles. The grouping is done with respect to the way the software is structured as a set of implementation units, as a set of elements that have responsibility at runtime and related interactions, and the way the software is related to nonsoftware structures in its environment, respectively [4]. In our work, we took all the three types of architectural style into consideration. The reason why we did so is that our main concern in the context of SPL testing is application architecture to reference architecture conformance analysis, i.e. we deal with SPL testing in architectural level. We tried to apply our approach to common and important architectural viewpoints such as decomposition viewpoint, uses viewpoint, generalization viewpoint and layered viewpoint in Module category, deployment viewpoint in Allocation category, and pipe-and-filter viewpoint in Component-and-connector category. However, it is important to note that our approach is applicable to use for all other viewpoints.

In the following sections we provide clear-cut information about the software architecture viewpoint, the metamodels considering the reference architecture

and application architecture, the architecture conformance comparison criteria (ACCC) specific to the software architecture viewpoint, and detailed steps of the comparison process.

5.1 Decomposition Viewpoint

This type of viewpoint expresses the system as elements of modules and submodules. In addition, it demonstrates how system responsibilities are distributed to these elements. Its relation is called Decomposition relation, which is a sort of the is-part-of relation. Moreover, it has two constraints as follows:

- No loops are allowed such as if module A contains module B, then module B cannot contain module A.
- There is at most one parent for a module. So, if module A contains module B, then module C cannot contain module B.

In our approach, we combine these two constraints into one that is no element can have the same name. This single constraint prevents the first constraint in a way that if module A contains module B, then module B cannot contain module A since there cannot be another module with the name "A". Likewise, the second constraint is prevented in the same way. We implemented this constraint in EVL as a validation rule in our approach's process of conformance checking of metamodel-model pairs.

The decomposition viewpoint metamodel for the reference architecture can be seen in Figure 4.2. The metaclasses called "Model, Element, Module, Subsystem, Property" compose the necessary metamodel that is based on the metamodel given in the work [53][54][55]. The followings can be inferred by looking the metamodel:

- Model consists of exactly one common part and optionally a variable part.

- Common part consists of at least one element.
- If variable part exists and has a variant, then the variant consists of exactly one element.
- Element, which has a name attribute, can have subelements.
- If there is a subelement, then there is a corresponding one parent element.
- Element is either a module or a subsystem.
- Element can have properties that have a name attribute and a value attribute.

The decomposition viewpoint metamodel for the application architecture can be seen in Figure 5.1, which is based on [53] as well. The decomposition viewpoint metamodels for the reference architecture and the application architecture are similar since in a SPL, the latter is derived from the former. Therefore, it is easy to infer same things by examining Figure 5.1 as the things we obtained from Figure 4.2. However, there is only one difference, considering only the metaclasses called "Model, Element, Module, Subsystem, Property" in Figure 4.2, that is "Model" is directly connected to "Element" since the application architecture does not have any information related to variability that can only be given in the reference architecture of a SPL.

The ACCC for the decomposition viewpoint can be listed as follows:

- Does every element of the reference architecture, which is listed as common, exist in the application architecture?
- Does every element of the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?

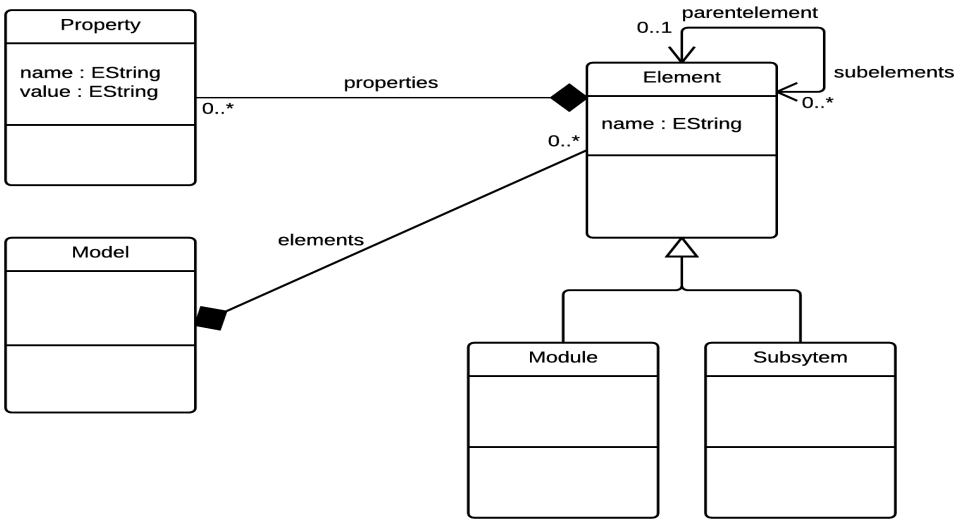

Figure 5.1: Decomposition viewpoint metamodel for the application architecture

- Does every subelement of the reference architecture, which is listed as common or variant, exist under the corresponding element in the application architecture, i.e. parent element of the subelement is correct?

Process of checking conformance of the two decomposition viewpoints, which are related to the reference architecture and application architecture, is demonstrated in Figure 5.2. The process has four major steps in which the ACCC is covered. It runs for each element of the application architecture and begins with checking for Common Convergence and Common Absence positions (step 1). After the candidates are marked for these two positions, the process continues to search for delta elements. If it finds any delta element in the application architecture, then this element is marked and saved for Delta position (step 2). The next step in the process is to compare elements for Variant Convergence and Variant Absence positions if there is at least one element in the application architecture, which is given as a variant in the reference architecture (step 3). In addition, if there is not any the application architecture's element that is given as a variant in the reference architecture, then the step 4 is not operated and the process is completed. The last step is related to Variant Convergence and Variant Absence positions, in which the process firstly checks whether there is a given variability

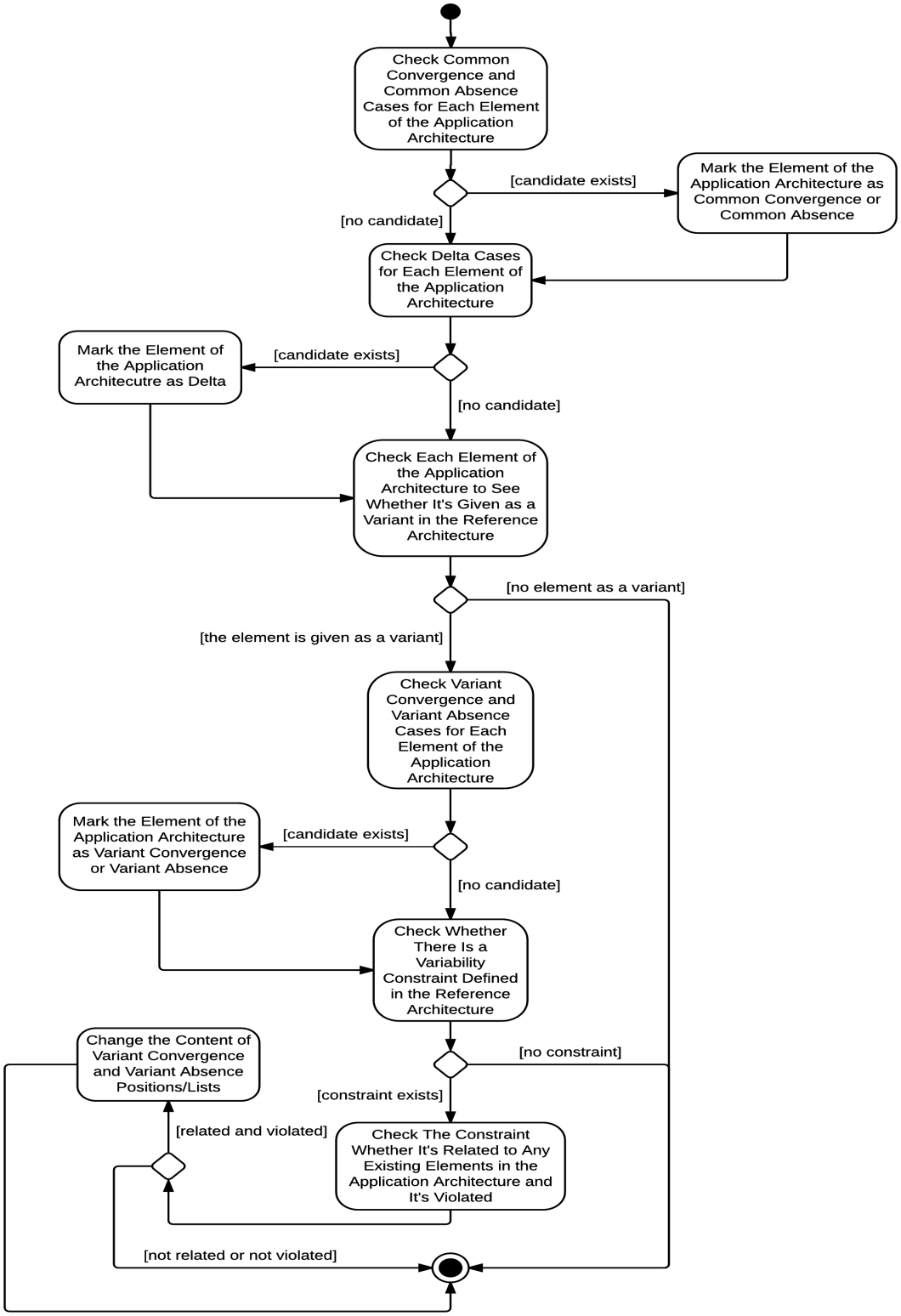

Figure 5.2: Main steps of comparison process

constraint for the variants or variation points in the reference architecture such as "V_E_VP", "VP_R_VP", etc. and if there is not any, then this step is not operated and the process is completed. Otherwise, the process checks whether the variability constraint is related to any existing elements in the application architecture and is violated in the application architecture. According to the verdict, it changes, i.e. adds or removes element, the content of Variant Convergence and Variant Absence positions/lists or does nothing (in case of it is not violated or is not related to any existing elements in the application architecture) (step 4), and the process is completed. The minor steps of the process can be listed as:

- *Finding Common Convergence and Common Absence Elements in Step 1*: Since in a SPL it is a must that all elements given as common in the reference architecture must be exist exactly, i.e. with the same name, type (module or subsystem for the decomposition viewpoint) and same parent element it has if there is any, in the application architecture, we check whether all common elements are given in the application architecture decomposition viewpoint model. The given elements are marked as Common Convergence and the others are marked as Common Absence.

- *Finding Delta Elements in Step 2*: We mark an element given in the application architecture as Delta if it has a different type (e.g. Module) and name (e.g. A) combination (e.g. Module A) from the all elements given in the reference architecture. A delta element is treated, in our approach, as it is a brand new element that does not exist in the reference architecture.

- *Finding Variant Convergence and Variant Absence Elements in Step 3*: While our tool is reading the two XML model files, it keeps track of the realized variation points of the reference architecture considering the elements given in the application architecture. Considering the type of relations the realized variation points have, the tool does followings:

  - If there is a mandatory relation in the realized variation point, the corresponding variant element must exist in the application architecture, which is a Variant Convergence case; otherwise, it is a Variant Absence

element. It is highly important to note that the tool, in this step, ignores any variability constraint that includes that variation point or that variation point's variant elements. The variability constraints, if there are any in the reference architecture, are considered in the Step 4 in which the content of the Variant Convergence and Variant Absence lists may be changed.

  - If there is an optional relation in the realized variation point, i.e. the realized variation point has an optional variant and it exists in the application architecture decomposition viewpoint model, then the variant element is marked as Variant Convergence. It is marked as Variant Absence if it does not exist in the application architecture decomposition viewpoint model although it is required by a variant or variation point included in a variability constraint given in the reference architecture, which is a case considered in Step 4.
  - If there is an alternative relation in the realized variation point, the tool firstly checks the constraint related to the alternative relation's attributes "min" and "max", which is mentioned in Section 4.2. If there is a violation of that constraint, all variant elements of the alternative relation are marked as Variant Absence. Otherwise, considering the constraint related to the values of "min" and "max" attributes, the tool checks that enough number of variant elements of the alternative relation exists in the application architecture decomposition viewpoint model: if it is the case, then the elements are marked as Variant Convergence.

- *Checking Defined Variability Constraints in Step 4*: The variability constraints in a SPL are basically two types that are "requires" or "excludes". If a variation point is required by a variant or another variation point, then the required variation point must be realized in the application architecture. Realization of a variation point means that at least one of its variant elements must exist in the application architecture. Likewise, if a variant is required by another variant, then the required variant must exist in the application architecture. The "excludes" variability constraints are

just the opposite of the "required" variability constraints, which means that the excluded variation point should not be realized or the excluded variant should not exist in the application architecture. Considering this explicit information, the tool firstly identifies the variability constraint type (e.g. "V_R_V", "VP_E_VP", etc.) and required/excluded part (i.e. variant or variation point). Then, it checks whether there is a violation of the variability constraint in the application architecture decomposition viewpoint model. In case of violation, the tool updates the lists of Variant Convergence and Variant Absence. In case of not violation, the tool does nothing.

- *Finding Not Selected Variants*: If an element is not marked in steps 1, 2, 3 or 4, then it must be marked as Not Selected Variant.

5.2 Uses Viewpoint

The uses viewpoint is built on top of the decomposition viewpoint, which describes how the modules of the system depend on each other [4]. The relation the uses viewpoint has called Uses relation, which is a kind of the depends-on relation. If module X uses module Y, then it means that in order to X satisfies its own requirements there must be the presence of a correctly functioning Y [4]. In addition, as it is stated in [4], there is no defined constraint for the uses viewpoint.

The uses viewpoint metamodel [53] for the reference architecture can be seen in Figure 5.3. The metaclasses related to the variability of the SPL are same as the ones given in Figure 4.2. Therefore, we provide explanation only for the metaclasses related to the uses viewpoint, which are as follows:

- Model consists of exactly one common part, optionally a variable part and at least one uses relation.
- Common part consists of at least one element.

- If variable part exists and has a variant, then the variant consists of exactly one element.
- Element, which has a name attribute, is either a module or a subsystem.
- Element can have properties that have a name attribute and a value attribute.
- Uses relation has one source and one target element in a way that the source uses the target.

The uses viewpoint metamodel for the application architecture can be seen in Figure 5.4 [53], which is exactly same as the one for the reference architecture without considering the metaclasses representing the variability demonstrated in Figure 5.3.

The ACCC for the uses viewpoint can be listed as follows:

- Does every source element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every target element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every source element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every target element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every uses relation given in the reference architecture exist in the application architecture?

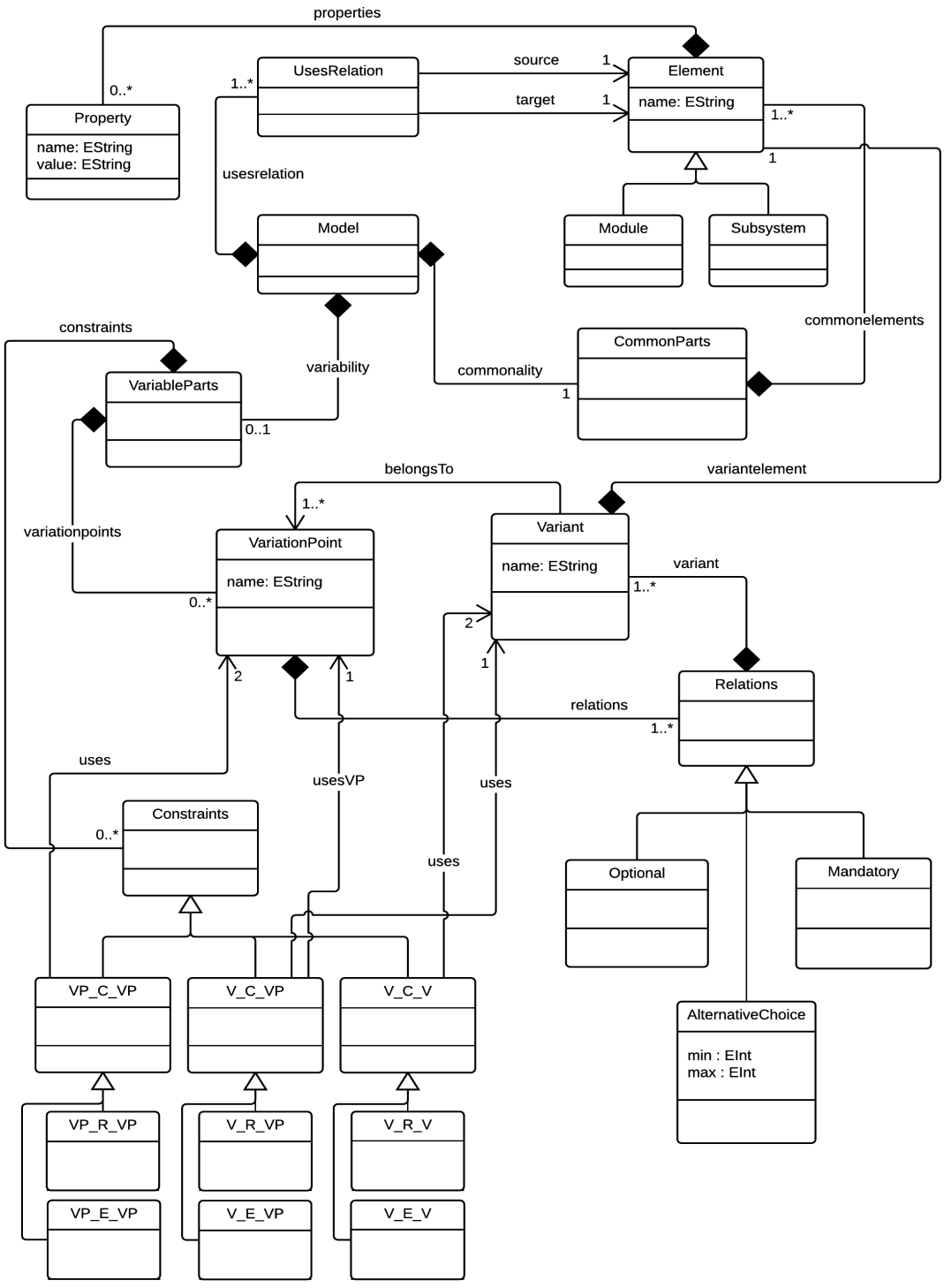

Figure 5.3: Uses viewpoint metamodel for the reference architecture

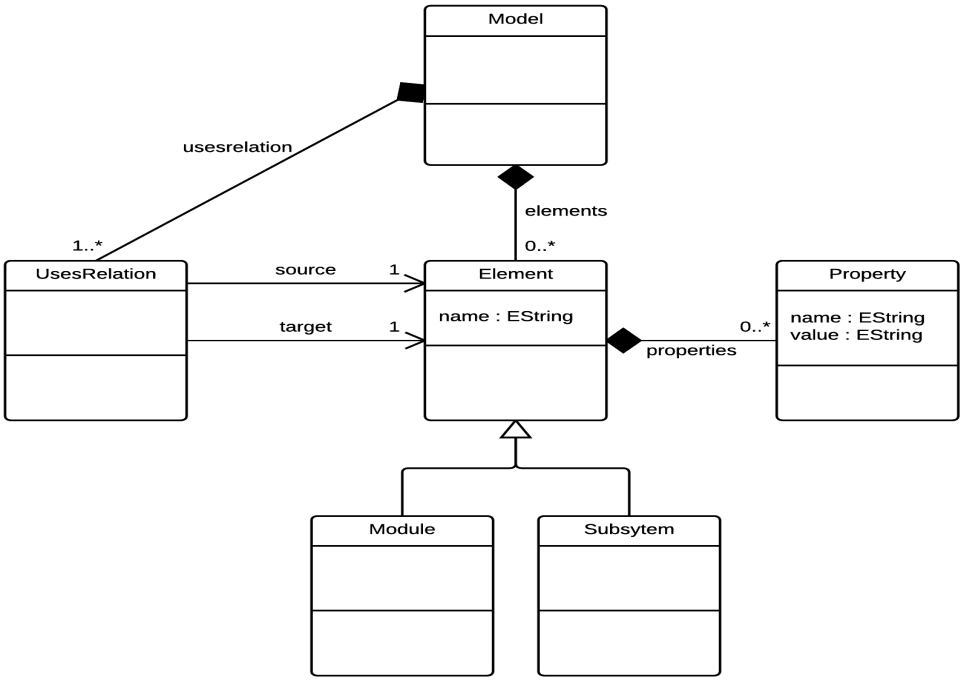

Figure 5.4: Uses viewpoint metamodel for the application architecture

In the conformance checking process of our approach for the uses viewpoint, we assume, as it is stated in [4], that if the source element of the uses relation contains subelements (module or subsystem), then there is at least one subelement in the source element, which is depended on the target element of that uses relation. Otherwise, the decomposition is not consistent, which then means that the uses relation definition in the input model is not correct; hence, the comparison result might be wrong. In addition, we assume that all uses relations given in the input model are consistent with the model elements exist in the input model. If the model input is only composed of module A and B, then there cannot be a uses relation between module A and C that does not exist in the input model, for example. Otherwise, this would be an inconsistency between the decomposition viewpoint and uses viewpoint of the same input model.

The reflexion model generated as a result of the architecture conformance analysis of the uses viewpoints contains information about the status of the uses

relations given in the input viewpoint models. This status can either be Convergence, Absence, Delta or Not Selected Uses Relation. The logic behind how this information is decided is as follows:

- If the source or target element of the uses relation given in the application architecture is a delta element (see Section 5.1), then any uses relation either is included is directly Delta type. Moreover, if a uses relation exists between two Convergence elements in the application architecture but does not exist in the reference architecture, then it is Delta type.
- If the source element of the uses relation given in the reference architecture is Not Selected Variant type (see Section 5.1), then the uses relation is Not Selected Uses Relation type since the uses relation without having a valid source element is meaningless.
- If the source element of the uses relation given in the reference architecture is Common Absence or Variant Absence type (see Section 5.1), then the uses relation is Absence type since the elements marked as Common Absence or Variant Absence must exist in the application architecture; hence, the uses relation includes those elements as the source must exist in the application architecture as well.
- If the source element of the uses relation given in the reference architecture is selected in the application architecture but the target element is not, then the uses relation is Absence type.
- If both the elements of the uses relation given in the reference architecture are selected in the application architecture, then
  - If the elements of the uses relations are exactly same, then it is Convergence type. Otherwise, it is Absence type.

Process of checking conformance of the two uses viewpoints, which are related to the reference architecture and application architecture, has two major steps in

which the ACCC is covered. It runs for each element of the application architecture and starts with the comparison process of the decomposition viewpoint given in Section 5.1 except considering the step of checking the parent element of each model element since in the uses viewpoint metamodel there is no subelement-parent element relation, which is also not necessary for uses viewpoint models [4]. After this step, the algorithm of finding the status of each uses relation in the uses viewpoint models, which is stated above, is run as the second step.

5.3 Generalization Viewpoint

The generalization viewpoint is useful to assist extension and evolution of software architectures and particular elements [4]. The relation it employs is called Generalization relation, which is based on the is-a relation. The viewpoint states that if a child module is-a parent module, then the parent module has commonalities the child module inherits and the variations are realized in the child module [4]. There is one constraint defined for the viewpoint, which underlines that in the Generalization relation cycles between parent and child modules are not allowed. For example, if module A is-a module B, then it is not possible to say that module B is-a module A in the same viewpoint model. Likewise, it is a violation of the constraint that having a module A is-a module A in the model. We implemented this constraint in EVL as a validation rule in our approach's process of conformance checking of metamodel-model pairs.

The generalization viewpoint metamodel [53] for the reference architecture can be seen in Figure 5.5. The metaclasses related to the variability of the SPL are same as the ones given in Figure 4.2. Therefore, we provide explanation only for the metaclasses related to the generalization viewpoint, which are as follows:

- Model consists of exactly one common part, optionally a variable part and at least one generalization relation.
- Common part consists of at least one element.

- If variable part exists and has a variant, then the variant consists of exactly one element.
- Element, which has a name attribute, is either a module or an interface.
- Generalization relation is either Class Inheritance that has a parent module and child module, Interface Implementation that has a parent interface and child module, or Interface Inheritance that has a parent interface and child interface. In addition, in this relation, the parent is the one being generalized and the child is the one generalizes the parent.

The generalization viewpoint metamodel for the application architecture can be seen in Figure 5.6 [53], which is exactly same as the one for the reference architecture without considering the metaclasses representing the variability demonstrated in Figure 5.5.

The ACCC for the generalization viewpoint can be listed as follows:

- Does every child element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every parent element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every child element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every parent element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every generalization relation that can be Class Inheritance, Interface Implementation or Interface Inheritance given in the reference architecture exist in the application architecture?

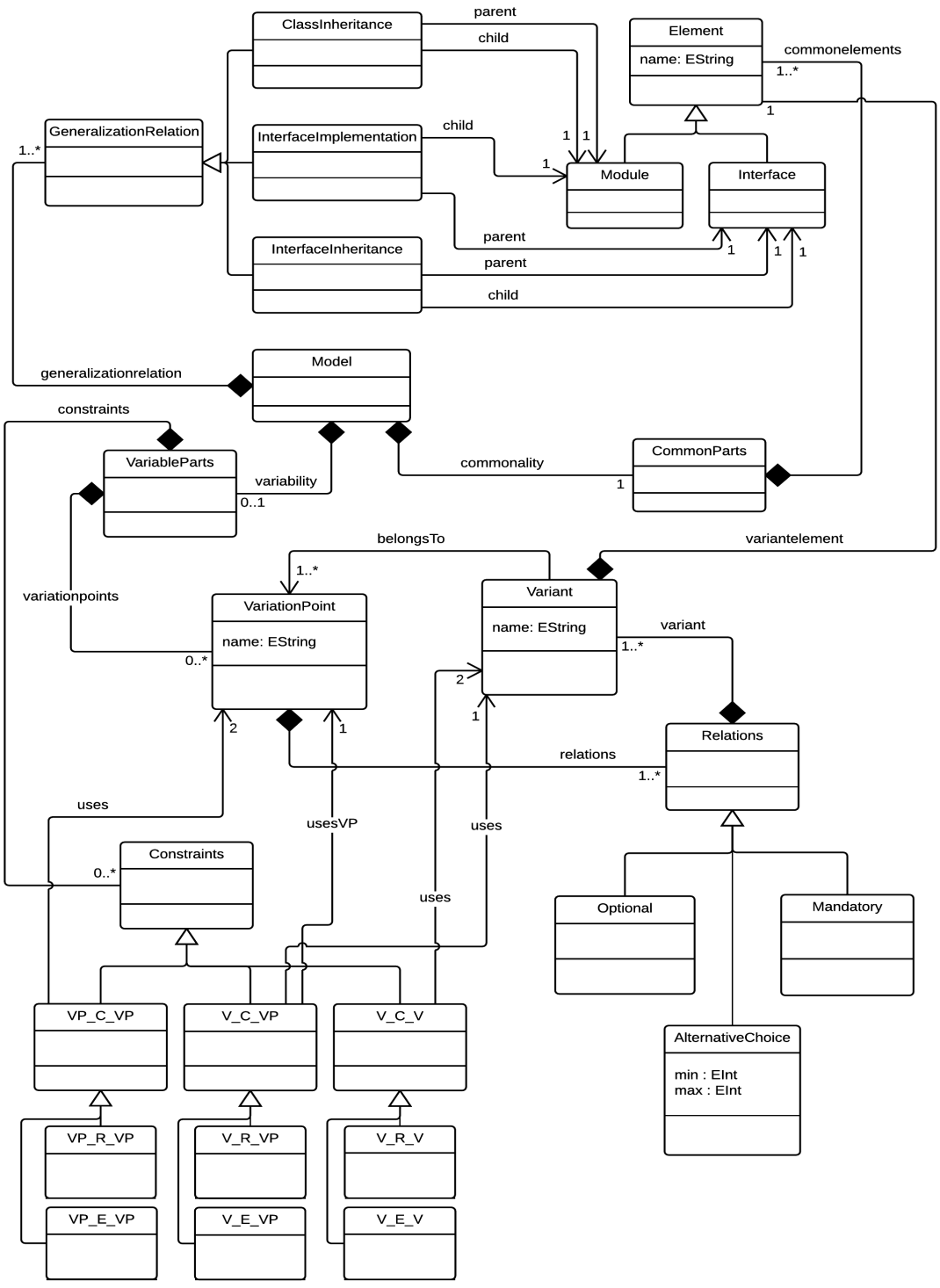

Figure 5.5: Generalization viewpoint metamodel for the reference architecture

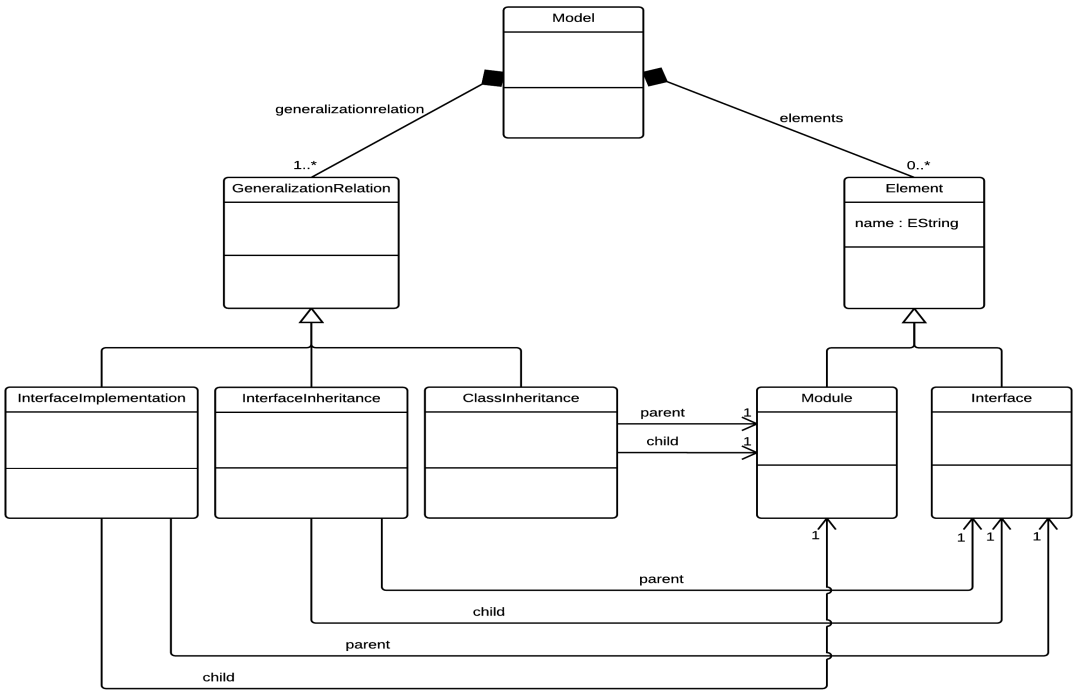

Figure 5.6: Generalization viewpoint metamodel for the application architecture

In the conformance checking process of our approach for the generalization viewpoint, we assume that each given generalization viewpoint model has generalization relations in a way that their parent and child element are not null (e.g. null is-a module A or null is-a null are not allowed in the input models).

The reflexion model generated as a result of the architecture conformance analysis of the generalization viewpoints contains information about the status of the generalization relations given in the input viewpoint models. This status can either be Convergence, Absence, Delta or Not Selected Generalization Relation. The logic behind how this information is decided is as follows:

- If the child or parent element of the generalization relation given in the application architecture is a delta element (see Section 5.1), then any generalization relation either is included is directly Delta type. Moreover, if a generalization relation exists between two Convergence elements in the application architecture but does not exist in the reference architecture, then

it is Delta type.

- If the child element of the generalization relation given in the reference architecture is Not Selected Variant type (see Section 5.1), then the generalization relation is Not Selected Generalization Relation type since the generalization relation without having a valid child element is meaningless.

- If the child element of the generalization relation given in the reference architecture is Common Absence or Variant Absence type (see Section 5.1), then the generalization relation is Absence type since the elements marked as Common Absence or Variant Absence must exist in the application architecture; hence, the generalization relation includes those elements as the child must exist in the application architecture as well.

- If the child element of the generalization relation given in the reference architecture is selected in the application architecture but the parent element is not, then the generalization relation is Absence type.

- If both the elements of the generalization relation given in the reference architecture are selected in the application architecture, then

  - If the elements of the generalization relations are exactly same, then it is Convergence type. Otherwise, it is Absence type.

Process of checking conformance of the two generalization viewpoints, which are related to the reference architecture and application architecture, has two major steps in which the ACCC is covered. It runs for each element of the application architecture and starts with the comparison process of the decomposition viewpoint given in Section 5.1 except considering the step of checking the parent element of each model element since in the generalization viewpoint metamodel there is no subelement-parent element relation, which is also not necessary for generalization viewpoint models [4]. After this step, the algorithm of finding the status of each generalization relation in the generalization viewpoint models, which is stated above, is run as the second step.

5.4 Layered Viewpoint

The layered viewpoint is used to mirror a partition of the software into units that are, in this case, layers [4]. Each of them demonstrates a set of modules that provides a related group of services. There is a similarity between uses viewpoint and layered viewpoint in terms of the relation they have. In the layered viewpoint, layers have a unidirectional relation between each other called Allowed-to-use, which is a type of depends-on relation as Uses relation. As it is stated in [53] that if layer A is allowed to use layer B, then the implementation of layer A can use any public modules of layer B. However, the opposite is not allowed. The viewpoint has three constraints such as every part of software is allocated to exactly one layer, there must be at least two layers, and the allowed-to-use relations should not be circular [4]. Although we do not have to check these constraints in our tool since we deal with only architecture conformance analysis of given two viewpoint models that already conform to these constraints, we provide EVL validation rules for them.

In a layered viewpoint, it is possible to show the modules inside a layer explicitly, i.e. the modules' name, the relation between them, etc. are demonstrated. It is also possible to depict a layered viewpoint only including layers with their name and Allowed-to-use relations. In our work, we assume that input layered viewpoint models are given according to later case; hence, in the comparison step of our approach, we do not deal with module information that exists in the layers.

The layered viewpoint metamodel [53] for the reference architecture can be seen in Figure 5.7. The metaclasses related to the variability of the SPL are same as the ones given in Figure 4.2. Therefore, we provide explanation only for the metaclasses related to the layered viewpoint, which are as follows:

- Model consists of exactly one common part, optionally a variable part and at least one layer relation.
- Common part consists of at least one layer.

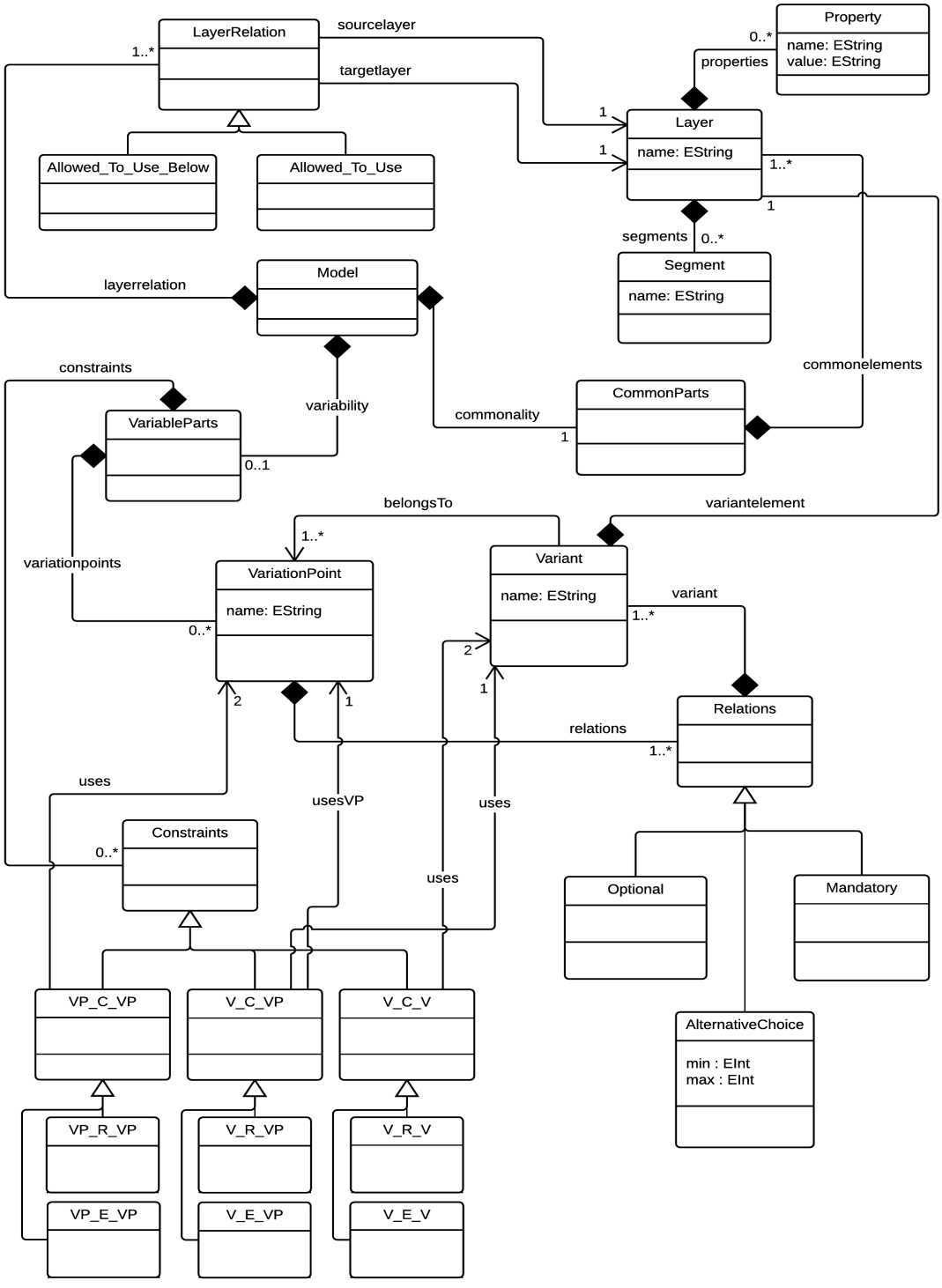

Figure 5.7: Layered viewpoint metamodel for the reference architecture

- If variable part exists and has a variant, then the variant consists of exactly one layer.
- Layer can have properties, which have a name attribute and a value attribute, and segments that have a name attribute.
- Layer relation has one source and one target layer in a way that the source is allowed to use the target. In addition, layer relation is type of either "allowed to use below" or "allowed to use". The former indicates "the source layer is allowed to use the target layer and all layers below it in the layering hierarchy" [53].

The layered viewpoint metamodel for the application architecture can be seen in Figure 5.8 [53], which is exactly same as the one for the reference architecture without considering the metaclasses representing the variability demonstrated in Figure 5.7.

The ACCC for the layered viewpoint can be listed as follows:

- Does every source layer in the reference architecture, which is listed as common, exist in the application architecture?
- Does every target layer in the reference architecture, which is listed as common, exist in the application architecture?
- Does every source layer in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every target layer in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?

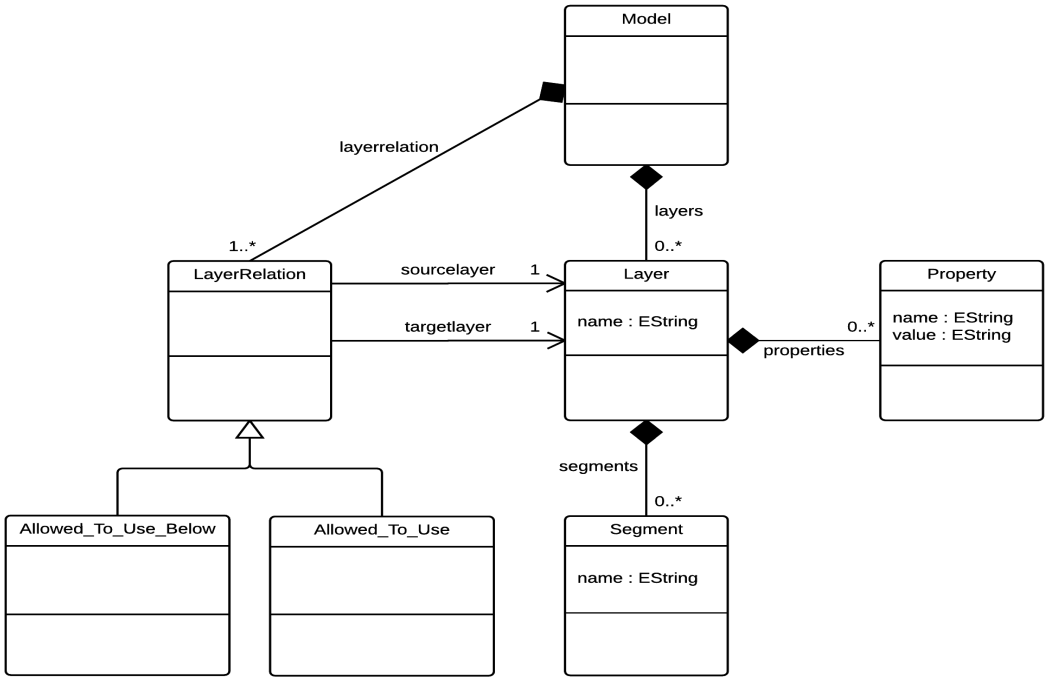

Figure 5.8: Layered viewpoint metamodel for the application architecture

- Does every layer relation that can be Allowed-to-use-below or Allowed-to-use given in the reference architecture exist in the application architecture?

The reflexion model generated as a result of the architecture conformance analysis of the layered viewpoints contains information about the status of the layer relations given in the input viewpoint models. This status can either be Convergence, Absence, Delta or Not Selected Layer Relation. The logic behind how this information is decided is as follows:

- If the source or target layer of the layer relation given in the application architecture is a delta layer (see Section 5.1), then any layer relation either is included is directly Delta type. Moreover, if a layer relation exists between two Convergence layers in the application architecture but does not exist in the reference architecture, then it is Delta type. However, if the layer relation exists in the reference architecture as well with one difference that

the type of layer relation is different, then it is Absence type since each of the application architectures should conform the reference architecture.

- If the source layer of the layer relation given in the reference architecture is Not Selected Variant type (see Section 5.1), then the layer relation is Not Selected Layer Relation type since the layer relation without having a valid source layer is meaningless.

- If the source layer of the layer relation given in the reference architecture is Common Absence or Variant Absence type (see Section 5.1), then the layer relation is Absence type since the layers marked as Common Absence or Variant Absence must exist in the application architecture; hence, the layer relation includes those layers as the source must exist in the application architecture as well.

- If the source layer of the layer relation given in the reference architecture is selected in the application architecture but the target layer is not, then the layer relation is Absence type.

- If both the layers of the layer relation given in the reference architecture are selected in the application architecture, then

  - If the layers of the layer relations are exactly same with having same type of layer relation such as Allowed-to-use-below or Allowed-to-use, then it is Convergence type.

Process of checking conformance of the two layered viewpoints, which are related to the reference architecture and application architecture, has two major steps in which the ACCC is covered. It runs for each layer (i.e. each element) of the application architecture and starts with the comparison process of the decomposition viewpoint given in Section 5.1 except considering the step of checking the parent element of each model element and their types (i.e. module or subsystem) since in the layered viewpoint metamodel there is no such relation and types for layers. After this step, the algorithm of finding the status of each layer relation in the layered viewpoint models, which is stated above, is run as the second step.

5.5 Deployment Viewpoint

Deployment viewpoint shows the mapping between the connectors and components of software system and the hardware of a computing platform on which the software system executes [4]. The main purpose of using deployment viewpoint is to analyze availability, performance, security and reliability of the software. Fundamental elements of this viewpoint are hardware elements, mainly called environmental elements, such as processor, memory, disk, etc. and software elements that are related to runtime entities such as threads, ports, processes, etc. Likewise, the main relation of the viewpoint is called Allocated-to relation that indicates which software elements reside on which physical units [4]. The allocation can dynamically change as well; hence, three different additional relations are defined in [4] such as Migrates-to, Copy-migrates-to and Execution-migrates-to. Lastly, there is no defined constraint for deployment viewpoint, as it is stated in [4].

The work [53] defines a metamodel for deployment viewpoint in which the Allocated-to relation is given implicitly by demonstrating that each hardware element includes software elements. Our metamodel for deployment viewpoint based on [53] can be seen in Figure 5.9. The metaclasses related to the variability of the SPL are same as the ones given in Figure 4.2. Therefore, we provide explanation only for the metaclasses related to the deployment viewpoint, which are as follows:

- Model consists of exactly one common part and optionally a variable part, and can include many migration relations and connections.
- Common part consists software elements and hardware elements, which have attributes name and type.
- Software element can have properties that have attributes name and value.
- Hardware element, which has attributes name and type, can have software elements so that an Allocated-to relation can be established. Likewise, it can have properties that have attributes name and value.

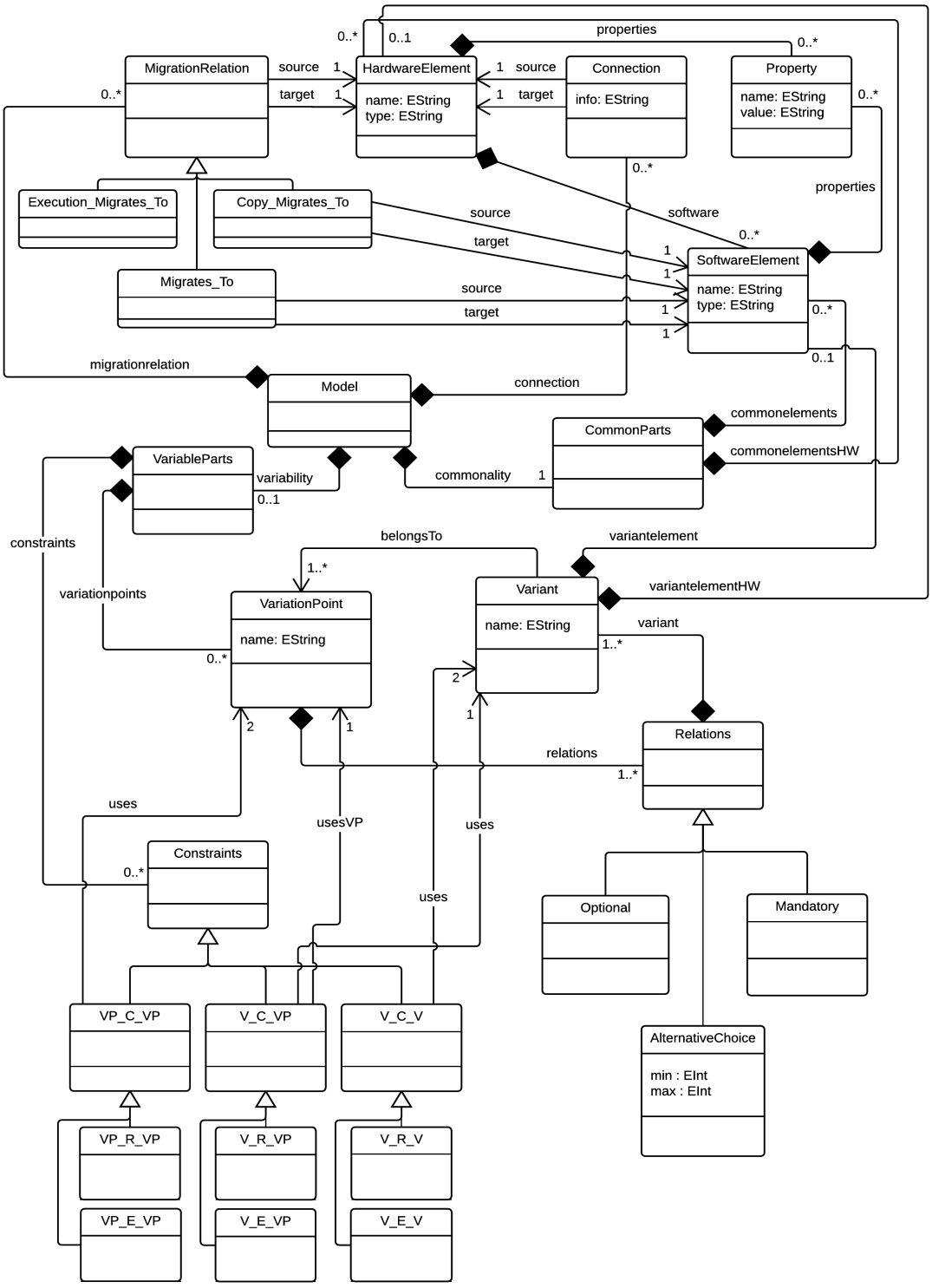

Figure 5.9: Deployment viewpoint metamodel for the reference architecture

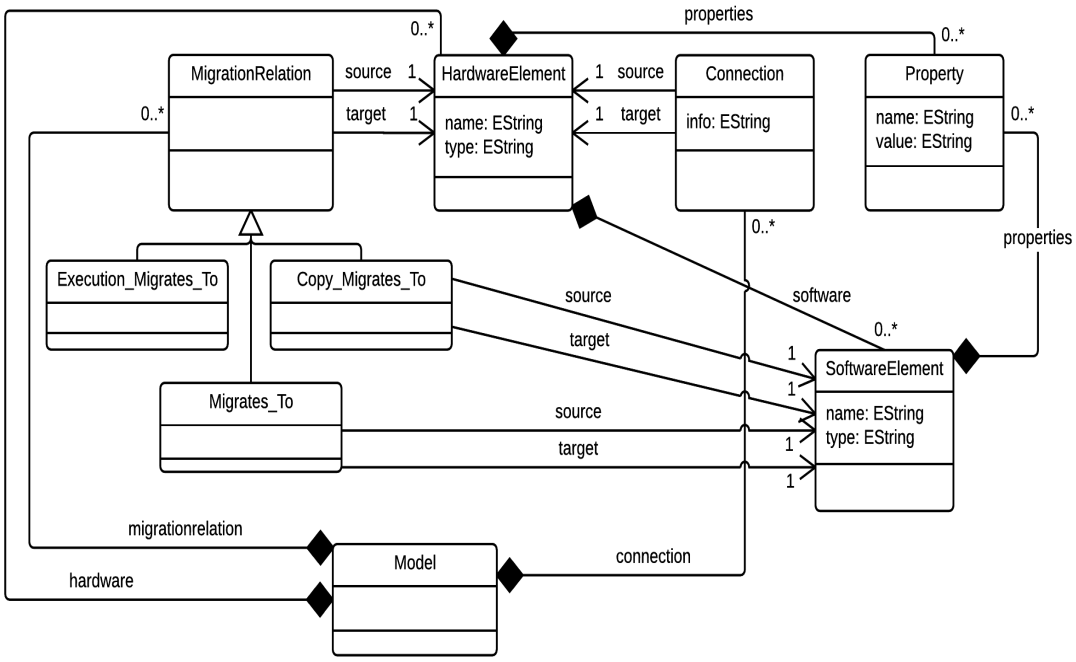

Figure 5.10: Deployment viewpoint metamodel for the application architecture

- A migration relation is either Execution-migrates-to, Copy-migrates-to or Migrates-to. In addition, a migration operation is completed between two hardware elements; hence, in the metamodel, there are two links that are from migration relation to hardware element to indicate source hardware element and target hardware element.
- Copy-migrates-to and Migrates-to relations use software element as source and target.
- There can be Connection links between hardware elements.

The deployment viewpoint metamodel for the application architecture can be seen in Figure 5.10 [53], which is exactly same as the one for the reference architecture without considering the metaclasses representing the variability demonstrated in Figure 5.9.

It is crucial to note that, in the deployment viewpoint, the variability is defined in terms of both software elements and hardware elements. In other words, the common and variable parts can have both software elements and hardware elements in this case.

The ACCC for the deployment viewpoint can be listed as follows:

- Does every hardware element of the reference architecture, which is listed as common, exist in the application architecture with having same connections between each other and consisting of same software elements (i.e. having same Allocated-to relations)?
- Does every hardware element of the reference architecture, which is listed as variant, exist in the application architecture with having same connections between each other and consisting of same software elements (i.e. having same Allocated-to relations), and considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every software element of the reference architecture, which is listed as common, exist in the application architecture with residing on same hardware element?
- Does every software element of the reference architecture, which is listed as variant, exist in the application architecture with residing on same hardware element and considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every migration relation that can be Execution-migrates-to, Migrates-to or Copy-migrates-to given in the reference architecture, if there is defined any, exist in the application architecture with having same source and target software elements?

The reflexion model generated as a result of the architecture conformance analysis of the deployment viewpoints contains information about the status of the deployment relations (Allocated-to, Migrates-to, Copy-migrates-to and Execution-migrates-to) and connection links between the hardware elements given in the

input viewpoint models. The status of deployment relations can either be Convergence, Absence, Delta or Not Selected Deployment Relation, and the status of connection links can either be Convergence, Absence, Delta or Not Selected Connection Link. The logic behind how this information is decided is as follows:

- If the source or target hardware element of the connection link given in the application architecture is a delta hardware element, then the connection link either is included is directly Delta type. A delta hardware element here means that combination of the element's name and type in the application architecture is different from combination of corresponding element's name and type in the reference architecture. Moreover, if a connection link exists between two Convergence hardware elements in the application architecture but does not exist in the reference architecture, then it is Delta type. However, if the connection link exists in the reference architecture as well with one difference that its info attribute is different, then it is Absence type.

- If the source hardware element of the connection link given in the reference architecture is Not Selected Variant type, then the connection link is Not Selected Connection Link type since the connection link without having a valid source hardware element is meaningless.

- If the source hardware element of the connection link given in the reference architecture is Absence type, then the connection link is Absence type since the hardware elements marked as Absence must exist in the application architecture; hence, the connection link includes those hardware elements as the source must exist in the application architecture as well.

- If the source hardware element of the connection link given in the reference architecture is selected in the application architecture but the target hardware element is Not Selected Variant type, then the connection link is Not Selected Connection Link type.

- If the source hardware element of the connection link given in the reference architecture is selected in the application architecture but the target

hardware element is not, then the connection link is Absence type.

- If both the hardware elements of the connection link given in the reference architecture are selected in the application architecture, then
  - If the hardware elements of the connection link are exactly same and the info attribute of the connection link is same, then it is Convergence type.
- If the source or target software/hardware element of the deployment relation given in the application architecture is a delta software/hardware element, then any deployment relation either is included is directly Delta type. A delta software/hardware element here means that combination of the element's name and type in the application architecture is different from combination of corresponding element's name and type in the reference architecture. In addition, if a deployment relation exists between two non-delta elements in the application architecture but does not exist in the reference architecture, then it is Delta type. However, if the deployment relation exists in the reference architecture as well with one difference that the type of deployment relation is different, then it is Absence type since each of the application architectures should conform the reference architecture.
- If the source software element of the deployment relation given in the reference architecture is Not Selected Variant type, then the deployment relation is Not Selected Deployment Relation type since the deployment relation without having a valid source software element is meaningless.
- If the source software/hardware element of the deployment relation given in the reference architecture is Absence type, then the deployment relation is Absence type since the software/hardware elements marked as Absence must exist in the application architecture; hence, the deployment relation includes those elements as the source must exist in the application architecture as well.
- If the source software/hardware element of the deployment relation given in the reference architecture is selected in the application architecture but

the target software/hardware element is Not Selected Variant type, then the deployment relation is Not Selected Deployment Relation type.

- If the source software/hardware element of the deployment relation given in the reference architecture is selected in the application architecture but the target software/hardware element is not, then the deployment relation is Absence type.

- If both the software/hardware elements of the deployment relation given in the reference architecture are selected in the application architecture, then

  – If the elements of the deployment relations are exactly same with having same type of deployment relation such as Allocated-to, Migrates-to, Copy-migrates-to or Execution-migrates-to, then it is Convergence type.

Process of checking conformance of the two deployment viewpoints, which are related to the reference architecture and application architecture, has two major steps in which the ACCC is covered. The first step is similar to the process done in the first comparison step of other viewpoint types in which the model elements are compared. After this step, the algorithm of finding the status of each deployment relation and connection link in the deployment viewpoint models, which is stated above, is run as the second step.

5.6 Pipe-and-Filter Viewpoint

Pipe-and-filter viewpoint is used to demonstrate successive transformations of streams of data in a software system [4]. There are two main elements in a pipe-and-filter viewpoint, which are filter and pipe. A filter's input port is responsible to receive data and its output port is responsible to send the data to the next filter through a pipe. The relation of the pipe-and-filter viewpoint is called Attachment relation that associates filter output ports with data-in roles of a pipe and filter input ports with data-out roles of pipes. As it is underlined in [4] that the

viewpoint has two main constrains such as pipes link output ports of a filter to input ports of another filter and linked filters must be on the same page regarding the type of data being transferred. The first constraint is guaranteed by the metamodel definition of the viewpoint, which is given below, and the second constraint is satisfied by EVL code in our tool, which can be found in Section 6.1.6.

The pipe-and-filter viewpoint metamodel [53] for the reference architecture can be seen in Figure 5.11. The metaclasses related to the variability of the SPL are same as the ones given in Figure 4.2. Therefore, we provide explanation only for the metaclasses related to the pipe-and-filter viewpoint, which are as follows:

- Model consists of exactly one common part and optionally a variable part.
- Common parts can be composed of both pipes and filters.
- A variant element can be both a pipe and a filter.
- A filter and a pipe can have properties that have name and value.
- A pipe has a name and linked to exactly two filters.
- A filter has a name and must have at least one port.
- A port has a name and data type and can either be input port or output port.
- A pipe's data-out role is linked to a filter's input port and its data-in role is linked to the filter's output port.

The pipe-and-filter viewpoint metamodel for the application architecture can be seen in Figure 5.12 [53], which is similar to the one for the reference architecture without considering the metaclasses representing the variability demonstrated in Figure 5.11.

The ACCC for the pipe-and-filter viewpoint can be listed as follows:

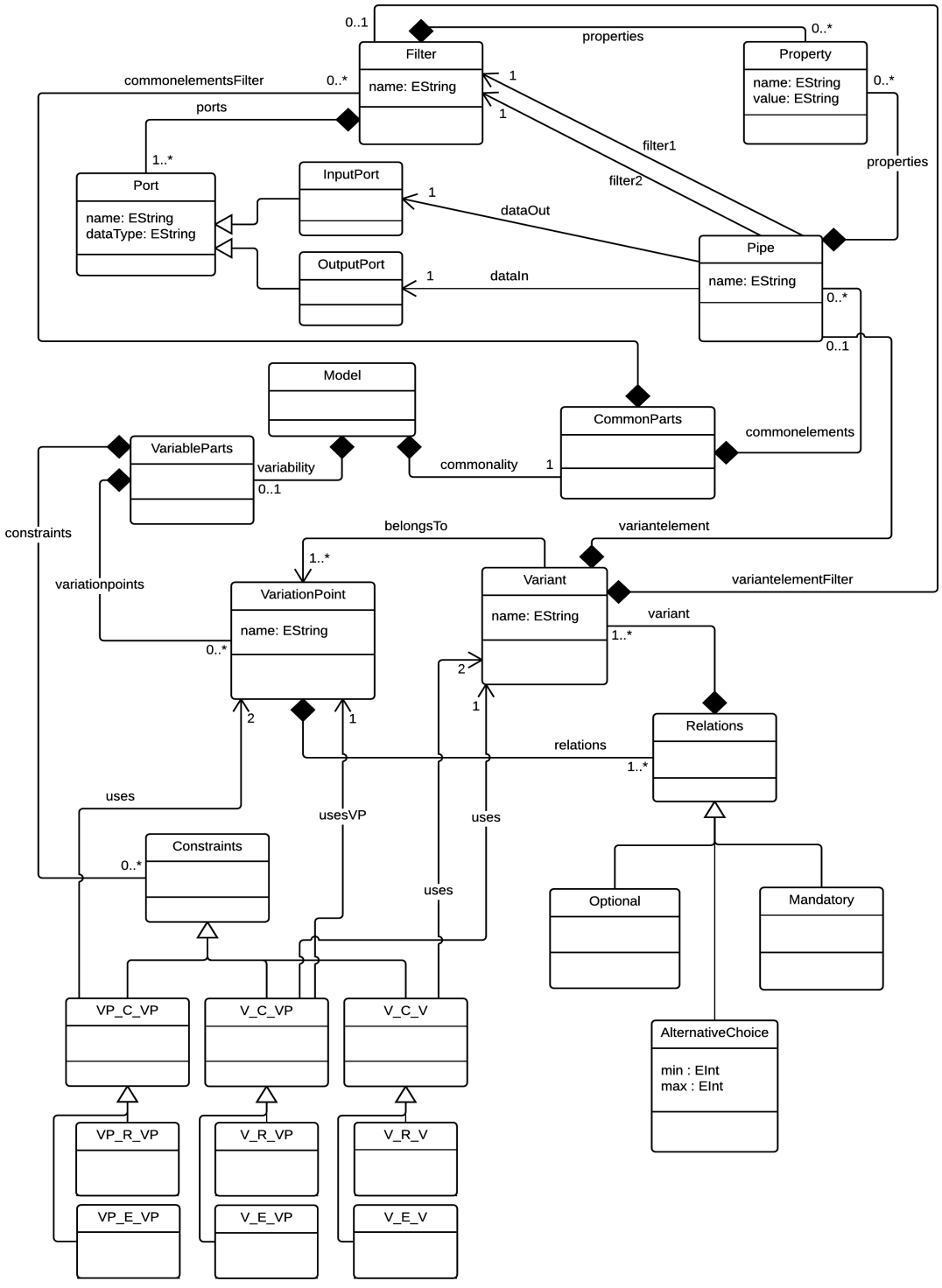

Figure 5.11: Pipe-and-filter viewpoint metamodel for the reference architecture

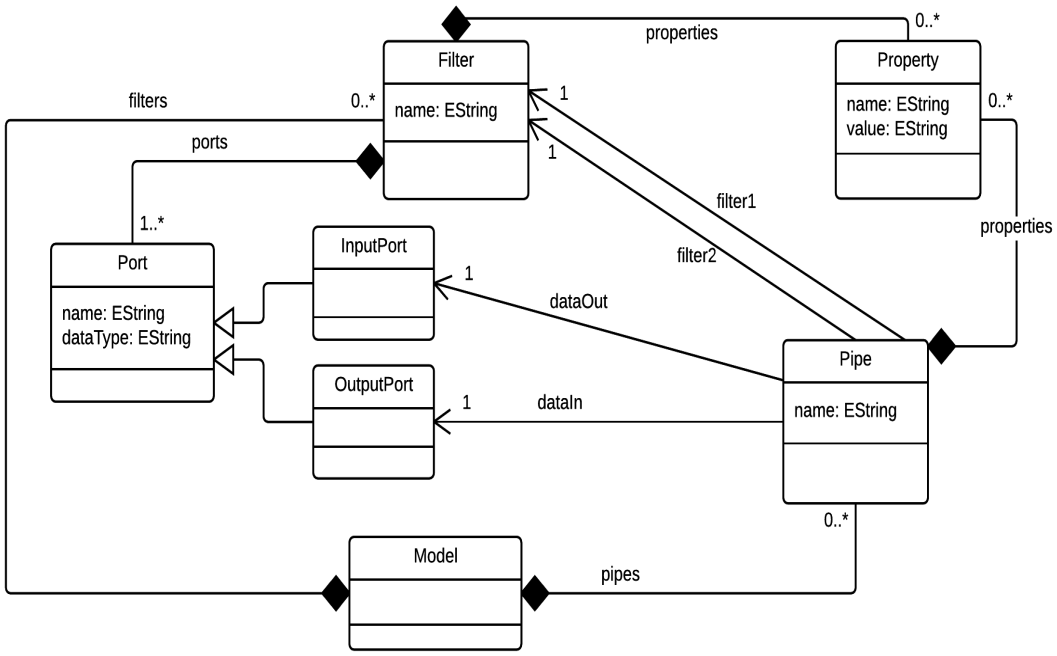

Figure 5.12: Pipe-and-filter viewpoint metamodel for the application architecture

- Does every filter element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every pipe element in the reference architecture, which is listed as common, exist in the application architecture?
- Does every filter element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every pipe element in the reference architecture, which is listed as variant, exist in the application architecture considering type of the relation the variant belongs to and restriction, if there exists any, the constraints impose?
- Does every attachment relation given in the reference architecture exist in the application architecture?

The reflexion model generated as a result of the architecture conformance analysis of the pipe-and-filter viewpoints contains information about the status of the Attachment relations given in the input viewpoint models. This status can either be Convergence, Absence, Delta or Not Selected Attachment Relation. The logic behind how this information is decided is as follows:

- If the filter or pipe element of the attachment relation given in the application architecture is a delta filter/pipe element, then any attachment relation either is included is directly Delta type. A delta filter/pipe element here means that the element's name and its ports (if it is a filter) in the application architecture are different from the corresponding element's name and its ports (if it is a filter) in the reference architecture. In addition, if an attachment relation exists between two Convergence elements in the application architecture but does not exist in the reference architecture, then it is Delta type.

- If the filter element of the attachment relation given in the reference architecture is Not Selected Variant type, then the attachment relation is Not Selected Attachment Relation type since the attachment relation without having a valid filter element is meaningless.

- If the filter element of the attachment relation given in the reference architecture is Absence type, then the attachment relation is Absence type since the filter elements marked as Absence must exist in the application architecture; hence, the attachment relation includes those filter elements must exist in the application architecture as well.

- If the filter element of the attachment relation given in the reference architecture is selected in the application architecture but the pipe element is Not Selected Variant type, then the attachment relation is Not Selected Attachment Relation type.

- If the filter element of the attachment relation given in the reference architecture is selected in the application architecture but the pipe element is Absence type, then the deployment relation is Absence type.

- If both the filter and pipe elements of the attachment relation given in the reference architecture are selected in the application architecture, then

  - If the elements of the attachment relations are exactly same, then it is Convergence type.

Process of checking conformance of the two pipe-and-filter viewpoints, which are related to the reference architecture and application architecture, has two major steps in which the ACCC is covered. The first step is similar to the process done in the first comparison step of other viewpoint types in which the model elements are compared. After this step, the algorithm of finding the status of each attachment relation in the pipe-and-filter viewpoint models, which is stated above, is run as the second step.

Chapter 6

Tool & Implementation

This chapter presents the subcomponents of our tool, their implementation details, a case study in which our approach was performed, and a discussion related to the performance of our approach and tool. We firstly give the subcomponents that are independent of the case study and listed as follows:

- Emfatic used to code metamodel definition
- EVL used to validate metamodel-model pairs
- EGL used to transform models into XML file (M2T transformation)
- Java code for model comparison process

6.1 The Subcomponents and the Software Architecture Viewpoints

6.1.1 Decomposition Viewpoint

The metamodels of the reference architecture and application architecture are generated using Emfatic, which are based on EMF. A snippet of the Emfatic code to generate the reference architecture decomposition viewpoint metamodel can be seen in Figure 6.1. The one for the application architecture decomposition viewpoint metamodel is similar to this one. The generated metamodels can be seen in Figure 4.2 and Figure 5.1.

In Figure 6.1, the lines 1-2 are specific to EMF file declaration. The first line declares the URI of the file with a prefix that is empty for this case. The line 4 declares a metaclass called "Model" with two composition relationships that are declared in the lines 5-6. The name after the keyword "val" is the name of the metaclass the relationship goes to. The number or question mark in the square brackets is about the multiplicity of the relationship. In lines 12-13 two opposite references are declared. The string after the "#" symbol indicates the opposite reference's name. The lines 17-18 have declaration of attributes type of "String". In the line 21 a class that extends another class is declared.

Validation process of the reference architecture decomposition viewpoint metamodel-model pair is done using EVL. In Figure 6.2, a snippet of EVL code for validating the reference architecture decomposition viewpoint metamodel-model pair is given. The EVL code given in the Figure 6.2 also checks the main constraint of the decomposition viewpoint (see line 11-15). The constraint declared in the line 6 checks whether each element of the model has a name. If the element has not a name, then the error is thrown. The lines 11-15 check the uniqueness of the name.

```
@namespace(uri="reference_architecture_decomposition_style_metamodel", prefix="")
package model;

class Model{
    val CommonParts[1] commonality;
    val VariableParts[?] variability;
}

class Element{
    attr String name;
    val Property[*] properties;
    ref Element[*]#parentelement subelements;
    ref Element[?]#subelements parentelement;
}

class Property{
    attr String name;
    attr String value;
}

class Module extends Element{}
```

Figure 6.1: Reference architecture decomposition viewpoint metamodel definition

```
// if the constraint is not met, then the message is displayed.

context Element{
    // The constrains of the decomposition style are handled here
    // All modules or subsystems must have a name
    constraint HasNameElement{
        check : self.name.isDefined()
        message : "The element " + self + " must have a name."
    }
    // All modules or subsytems must have a unique name
    constraint HasUniqueNameElement{
        guard : self.satisfies("HasNameElement")
        check : Element.allInstances.one(ele.name = self.name)
        message : "The name " + self.name + " is already in use in " + self
    }
    // No loops are allowed in parentelement-subelement relationship
    constraint NoLoop{
        check : not self.subelements.includes(self.parentelement)
        message: "Loop is found containing " + self +
        ", which is not allowed. Change the parentelement/subelement value."
    }
}
```

Figure 6.2: Validation code for reference architecture decomposition viewpoint

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Model>
3    <commonality>
4      [%if (CommonParts.all().commonelements.name.isDefined()){
5          for (names in CommonParts.all().commonelements){
6              for (namesC in names) {%]
7      <commonelements type="[%=namesC.type().name%]" name="[%=namesC.name%]"
8          [%if (namesC.properties.isDefined()){
9              for(pro in namesC.properties){%]
10       <properties name="[%=pro.name%]" value="[%=pro.value%]"/>
11             [%}%]
12         [%}%]
13         [%}
14     }
15 }%]
16   </commonality>
```

Figure 6.3: M2T transformation code for reference architecture decomposition viewpoint

EGL, which is used for M2T transformation in this work, operates on templates. It takes a model, which conforms a metamodel, as an input and based on the template prepared before, generates a text file. In Figure 6.3, a snippet of the template for the reference architecture decomposition viewpoint model, which is used to generate an XML file, is demonstrated. The strings in blue color are the static part of the template, which is directly written to the file. The strings between the symbols "[%" and "%]" are the dynamic part of the template and the values for this part change according to the instance of the input model. In line 4 the EGL checks whether the name of the common elements are given. If there is a name given for a common element, the loops in the lines 5-6 get the name that is later written to the file in line 7.

The Java code we designed for the architecture conformance analysis between the reference architecture decomposition viewpoint and application architecture decomposition viewpoint consists of 12 classes with more than 1700 lines of code; hence, it is not possible to demonstrate all of the code here. However, we provide in Figure 6.4 a snippet of the code, which checks Common Convergence and Common Absence elements. The code is self-explanatory with the comments it has, which basically does the step 1 of the process mentioned in Section 5.1.

The reflexion model generation process is handled by the Java code we designed, which creates two different forms that are image based and textual based. For the example reflexion models, please see Section 6.2.1.

```
// check common elements (common convergence + common absence is checked)
private static void isAllCommonElementsSame (ModelClass mc, ModelClassApp mca) {

    boolean isAdded = false;

    for (int i=0; i<mc.getCommonality().getCommonelements().size(); i++) {
        for (int j=0; j<mca.getElements().size(); j++) {
            // check Type + Name
            if (mc.getCommonality().getCommonelements().get(i).getType().
                    equals(mca.getElements().get(j).getType())
                    && mc.getCommonality().getCommonelements().get(i).getName().
                    equals(mca.getElements().get(j).getName())) {
                // check Parent element
                if (mc.getCommonality().getCommonelements().get(i).getParentelement() == null) {
                    if (mca.getElements().get(j).getParentelement() == null) {
                        isAdded = true;
                        break;
                    }else {
                        isAdded = false;
                    }
                }else {
                    if (mca.getElements().get(j).getParentelement() == null){
                        // the first one is not null but the second one is null, then they are different
                        isAdded = false;
                    }else {
                        // they are both not null, then check if they are same
                        if (mc.getCommonality().getCommonelements().get(i).getParentelement().
                                equals(mca.getElements().get(j).getParentelement())) {
                            isAdded = true;
                            break;
                        }else {
                            isAdded = false;
                        }
                    }
                }
            }else {
                isAdded = false;
            }
        }

        if (isAdded == true) {
            mcCommonConvergence.add(mc.getCommonality().getCommonelements().get(i));
        }else {
            mcCommonAbsence.add(mc.getCommonality().getCommonelements().get(i));
        }
    }
}
```

Figure 6.4: Code for finding elements of common convergence/absence

```
1  @namespace(uri="reference_architecture_uses_style_metamodel", prefix="")
2  package model;
3
4  class Model{
5      val CommonParts[1] commonality;
6      val VariableParts[?] variability;
7      val UsesRelation[+] usesrelation;
8  }
9
10 class UsesRelation{
11     ref Element[1] source;
12     ref Element[1] target;
13 }
14
15 class Element{
16     attr String name;
17     val Property[*] properties;
18 }
19
20 class Property{
21     attr String name;
22     attr String value;
23 }
24
25 class Module extends Element{}
```

Figure 6.5: Reference architecture uses viewpoint metamodel definition

6.1.2 Uses Viewpoint

A snippet of Emfatic code for metamodel definition of reference architecture uses viewpoint can be seen in Figure 6.5, which has only three differences from the one given for decomposition viewpoint. The first difference is that in class "Model" definition line 7 declares a composition relationship between the classes "Model" and newly declared "UsesRelation" that is the second difference. In "UsesRelation", there are two references to class "Element" to indicate source and target elements in the uses viewpoint model. The third difference is that the reference declarations of subelement and parent element in the class "Element" are removed. Emfatic code for the metamodel definition of application architecture uses viewpoint is similar to the one for reference architecture uses viewpoint that is demonstrated in Figure 6.5.

In Figure 6.6, a snippet of EVL code for validating the reference architecture uses viewpoint metamodel-model pair is given. The constraint defined in lines 17-28 is added to the EVL code developed for the decomposition viewpoint and responsible to check source and target elements of a uses relation defined in the uses model.

EGL template used in M2T transformation of the reference and application architecture uses viewpoints is based on the one used in the decomposition viewpoint. In Figure 6.7, EGL code used to write uses relations' source and target

```
17 context UsesRelation{
18     // Source element cannot be null
19     constraint NullSource{
20         check : self.source.isDefined()
21         message : "The source element " + self.source + " must be defined."
22     }
23     // Target element cannot be null
24     constraint NullTarget{
25         check : self.target.isDefined()
26         message : "The target element " + self.target + " must be defined."
27     }
28 }
```

Figure 6.6: Validation code for reference architecture uses viewpoint

```
50   [%if (Model.all.usesrelation.isDefined()){%]
51   <usesrelations>
52   [%for (ur in Model.all.usesrelation){
53   for (uR in ur){%]
54     <usesrelation source="[%if(uR.source.isDefined()){%][%=uR.source.name%][%}%]"
55     target="[%if(uR.target.isDefined()){%][%=uR.target.name%][%}%]"/>
56   [%}}%]
57   </usesrelations>
58   [%}%]
```

Figure 6.7: M2T transformation code for reference architecture uses viewpoint

elements into an XML file is demonstrated.

Figure 6.8 shows a self-explanatory snippet of the code for finding uses relations' status, in which delta status is checked, as it is explained in Section 5.2. Lastly, the generated reflexion model can be found in Section 6.2.2.

6.1.3 Generalization Viewpoint

A snippet of Emfatic code for metamodel definition of reference architecture generalization viewpoint can be seen in Figure 6.9. The lines between 10-25 declare the generalization relation with three types that are Class Inheritance, Interface Implementation and Interface Inheritance. Emfatic code for the metamodel definition of application architecture generalization viewpoint is similar to the one for reference architecture generalization viewpoint that is demonstrated in Figure 6.9.

In Figure 6.10, a snippet of EVL code for validating the reference architecture

```
994  private static void checkUsesRelations(ModelClass mc, ModelClassApp mca){
995      boolean isSame;
996      boolean isDelta;
997      // If the source or target element of the uses relation in the application architecture is a delta element,
998      // then any uses relation either is included is directly Delta type
999      for (int j=0; j<mca.getUsesrelation().size();j++) {
1000         if (mcaDelta.contains(mca.getUsesrelation().get(j).getSource()) ||
1001                 mcaDelta.contains(mca.getUsesrelation().get(j).getTarget())) {
1002             deltaUsesRelation.add(mca.getUsesrelation().get(j));
1003         }
1004     }
1005     // Moreover, if a uses relation exists in the application architecture
1006     // but does not exist in the reference architecture, then it is Delta type
1007     for (int i=0; i<mca.getUsesrelation().size(); i++) {
1008         // If both the elements are selected in the application architecture, then
1009         if ((isUsesRelationContained(mca.getUsesrelation().get(i).getSource(), mcCommonConvergence) ||
1010                 isUsesRelationContained(mca.getUsesrelation().get(i).getSource(), mcVariantConvergence)) &&
1011                 (isUsesRelationContained(mca.getUsesrelation().get(i).getTarget(), mcCommonConvergence) ||
1012                 isUsesRelationContained(mca.getUsesrelation().get(i).getTarget(), mcVariantConvergence))) {
1013             isDelta = true;
1014             for (int j=0; j<mc.getUsesrelation().size(); j++) {
1015                 if ((mc.getUsesrelation().get(j).getSource().getName()+""+
1016                         mc.getUsesrelation().get(j).getTarget().getName()).
1017                         equals(mca.getUsesrelation().get(i).getSource().getName()+""+
1018                         mca.getUsesrelation().get(i).getTarget().getName())) {
1019                     isDelta=false;
1020                     break;
1021                 }
1022             }
1023             if (isDelta == true){
1024                 deltaUsesRelation.add(mca.getUsesrelation().get(i));
1025             }
1026         }
1027     }
```

Figure 6.8: Code for finding status of uses relations

```
1  @namespace(uri="reference_architecture_generalization_style_metamodel", prefix="")
2  package model;
3
4  class Model{
5      val CommonParts[1] commonality;
6      val VariableParts[?] variability;
7      val GeneralizationRelation[+] generalizationrelation;
8  }
9
10 class GeneralizationRelation{}
11
12 class ClassInheritance extends GeneralizationRelation{
13     ref Module[1] parent;
14     ref Module[1] child;
15 }
16
17 class InterfaceImplementation extends GeneralizationRelation{
18     ref Interface[1] parent;
19     ref Module[1] child;
20 }
21
22 class InterfaceInheritance extends GeneralizationRelation{
23     ref Interface[1] parent;
24     ref Interface[1] child;
25 }
```

Figure 6.9: Reference architecture generalization viewpoint metamodel definition

```
17 context GeneralizationRelation{
18     // Parent element cannot be null
19     constraint NullParent{
20         check : self.parent.isDefined()
21         message : "The parent element " + self.parent + " must be defined."
22     }
23     // Child element cannot be null
24     constraint NullChild{
25         check : self.child.isDefined()
26         message : "The child element " + self.child + " must be defined."
27     }
28     // The constrain of the generalization style is handled here
29     // No loop allowed like A is-a A
30     constraint NoLoop1{
31         guard : self.satisfies("NullParent") and self.satisfies("NullChild")
32         check : self.parent <> self.child
33         message : "Loop is found for " + self.child.name + " is-a " + self.parent.name
34     }
35     // The constrain of the generalization style is handled here
36     // No loop allowed like A is-a B and B is-a A
37     constraint NoLoop2{
38         guard : self.satisfies("NullParent") and self.satisfies("NullChild")
39         check : not GeneralizationRelation.allInstances.one
40         (gr|gr.child.name = self.parent.name and gr.parent.name = self.child.name)
41         message : "Loop is found for " + self.child.name + " is-a " + self.parent.name
42     }
43 }
```

Figure 6.10: Validation code for reference architecture generalization viewpoint

generalization viewpoint metamodel-model pair is given. The codes in lines 30-34 and 37-42 check the constraint of generalization viewpoint stated in Section 5.3.

EGL template used in M2T transformation of the reference and application architecture generalization viewpoints is based on the one used in the decomposition viewpoint. In Figure 6.11, EGL code used to write generalization relations' child and parent elements with the generalization type into an XML file is demonstrated.

The code for finding the status of generalization relations is similar to one for uses relations that is partly depicted in Figure 6.8. In the code for uses relations, the comparison is mainly based on the model elements' name; however, in the code for generalization relations, it considers the model elements' type, which can be Module or Inheritance, as well as their name. Lastly, the generated reflexion model can be found in Section 6.2.3.

6.1.4 Layered Viewpoint

A snippet of Emfatic code for metamodel definition of reference architecture layered viewpoint can be seen in Figure 6.12. The lines 15 and 17 declare two types of layer relation called Allowed-to-use-below and Allowed-to-use via extension

```
[%if (Model.all.generalizationrelation.isDefined()){%]
<generalizationrelations>
[%for (ur in Model.all.generalizationrelation){
for (uR in ur){%]
  <generalizationrelation type="[%=uR.type().name%]" child="[%if(uR.child.isDefined()){%]
  [%=uR.child.name%][%}%]" parent="[%if(uR.parent.isDefined()){%][%=uR.parent.name%][%}%]"/>
[%}}%]
</generalizationrelations>
[%}%]
```

Figure 6.11: M2T transformation code for reference architecture generalization viewpoint

```
@namespace(uri="reference_architecture_layered_style_metamodel", prefix="")
package model;

class Model{
    val CommonParts[1] commonality;
    val VariableParts[?] variability;
    val LayerRelation[+] layerrelation;
}

class LayerRelation{
    ref Layer[1] sourcelayer;
    ref Layer[1] targetlayer;
}

class AllowedToUseBelow extends LayerRelation{}

class AllowedToUse extends LayerRelation{}

class Layer{
    attr String name;
    val Property[*] properties;
    val Segment[*] segments;
}

class Property{
    attr String name;
    attr String value;
}

class Segment{
    attr String name;
}
```

Figure 6.12: Reference architecture layered viewpoint metamodel definition

mechanism, respectively. Emfatic code for the metamodel definition of application architecture layered viewpoint is similar to the one for reference architecture layered viewpoint that is demonstrated in Figure 6.12.

In Figure 6.13, a snippet of EVL code for validating the reference architecture layered viewpoint metamodel-model pair is given. The codes in lines 3-10 and 12-25 partly check the constraint of layered viewpoint stated in Section 5.4.

In Figure 6.14, EGL code used to write layered relations' source and target layers with the layered relation type into an XML file is demonstrated.

The code for finding the status of layer relations is similar to one for uses relations that is partly depicted in Figure 6.8. As explained previously, in the code

```
3  context Model{
4      // The constrain of the layered style is handled here
5      // At least two layers must be defined. i.e. at least one layer relation must be defined
6      constraint AtLeastTwoLayersDefined{
7          check : self.layerrelation.size() >= 1
8          message : "At least two layers must be defined."
9      }
10 }
11
12 context Layer{
13     // All layers must have a name
14     constraint HasNameLayer{
15         check : self.name.isDefined()
16         message : "The layer " + self + " must have a name."
17     }
18     // The constrain of the layered style is handled here
19     // All layers must have a unique name
20     constraint HasUniqueNameLayer{
21         guard : self.satisfies("HasNameLayer")
22         check : Layer.allInstances.one(ele.name = self.name)
23         message : "The name " + self.name + " is already in use in " + self
24     }
25 }
```

Figure 6.13: Validation code for reference architecture layered viewpoint

```
61 [%if (Model.all.layerrelation.isDefined()){%]
62 <layerrelations>
63 [%for (ur in Model.all.layerrelation){
64 for (uR in ur){%]
65   <layerrelation type="[%=uR.type().name%]" source="[%if(uR.sourcelayer.isDefined()){%][%=uR.sourcelayer.name%][%}%]"
66   target="[%if(uR.targetlayer.isDefined()){%][%=uR.targetlayer.name%][%}%]"/>
67 [%}}%]
68 </layerrelations>
69 [%}%]
```

Figure 6.14: M2T transformation code for reference architecture layered viewpoint

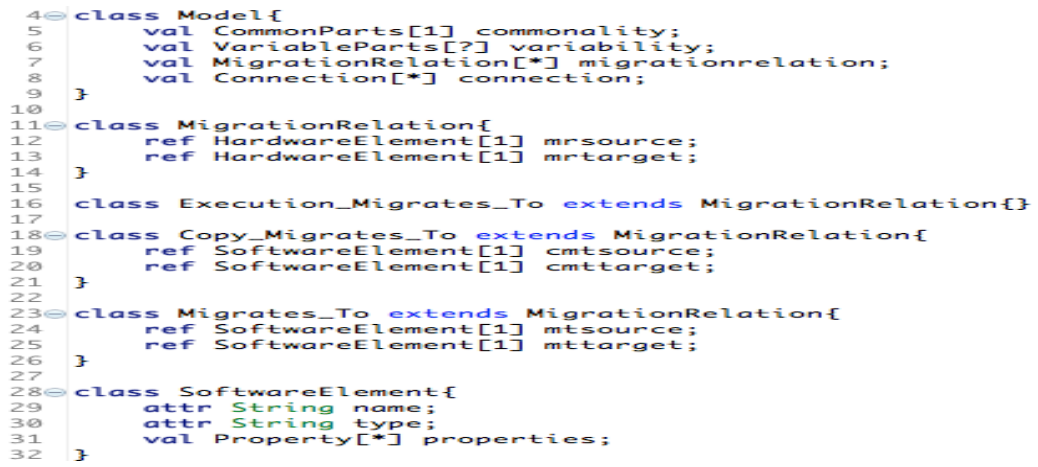

Figure 6.15: Reference architecture deployment viewpoint metamodel definition

for uses relations, the comparison is mainly based on the model elements' name; however, in the code for layer relations, it considers the layer relations' type, which can be Allowed-to-use-below or Allowed-to-use, as well as corresponding layers' name. Lastly, the generated reflexion models can be found in Section 6.2.4.

6.1.5 Deployment Viewpoint

A snippet of Emfatic code for metamodel definition of reference architecture deployment viewpoint can be seen in Figure 6.15. The lines between 11-14 are used to declare migration relation with source and target hardware element references. Likewise, the lines between 28-32 declare metaclass Software Element with its attributes called name and type. Emfatic code for the metamodel definition of application architecture deployment viewpoint is similar to the one for reference architecture deployment viewpoint that is demonstrated in Figure 6.15.

In Figure 6.16, a self-explanatory snippet of EVL code for validating the application architecture deployment viewpoint metamodel-model pair is given. The EVL code ensures that, which is also stated in [53], a hardware element cannot connect to itself, all types of migration relation must be established between two different hardware elements and the name of software element that is migrated to another hardware element via Copy-migrates-to or Migrates-to relation must

```
context SoftwareElement{
    // All software elements must have a name
    constraint HasNameElement{
        check : self.name.isDefined()
        message : "The software element " + self + " must have a name."
    }
    // All software elements must have a unique name
    constraint HasUniqueNameElement{
        guard : self.satisfies("HasNameElement")
        check : SoftwareElement.allInstances.one(ele.name = self.name)
        message : "The name " + self.name + " is already in use in " + self
    }
}

context Connection{
    // Source element cannot be null
    constraint NullSource{
        check : self.source.isDefined()
        message : "The source hardware element " + self.source + " must be defined."
    }
    // Target element cannot be null
    constraint NullTarget{
        check : self.target.isDefined()
        message : "The target hardware element " + self.target + " must be defined."
    }
}
```

Figure 6.16: Validation code for application architecture deployment viewpoint

```
[%if (Model.all().hardware.isDefined){%]
[%for (hard in Model.all().hardware){%]
[%for (so in hard){%]
[%if (so.software.isDefined){
for (soN in so.software){
for (soN1 in soN){%]
<allocatedto hardware="[%=so.name%]" software="[%=soN1.name%]"/>
[%}}}}%]
[%}%]
[%}%]
[%if (Connection.all().info.isDefined()) {
for(names in Connection.all()) {%]
<connections info="[%=names.info%]" source="[%=names.source.name%]" target="[%=names.target.name%]"/>
[%}}%]
```

Figure 6.17: M2T transformation code for application architecture deployment viewpoint

be same in that relation's source and target attributes.

In Figure 6.17, EGL code used to write connection links between hardware elements and Allocated-to relations' hardware and software elements into an XML file is demonstrated.

In the code for finding the status of deployment relations, the main idea is similar to one developed for uses relations that is partly depicted in Figure 6.8 and, lastly, the generated reflexion models can be found in Section 6.2.5.

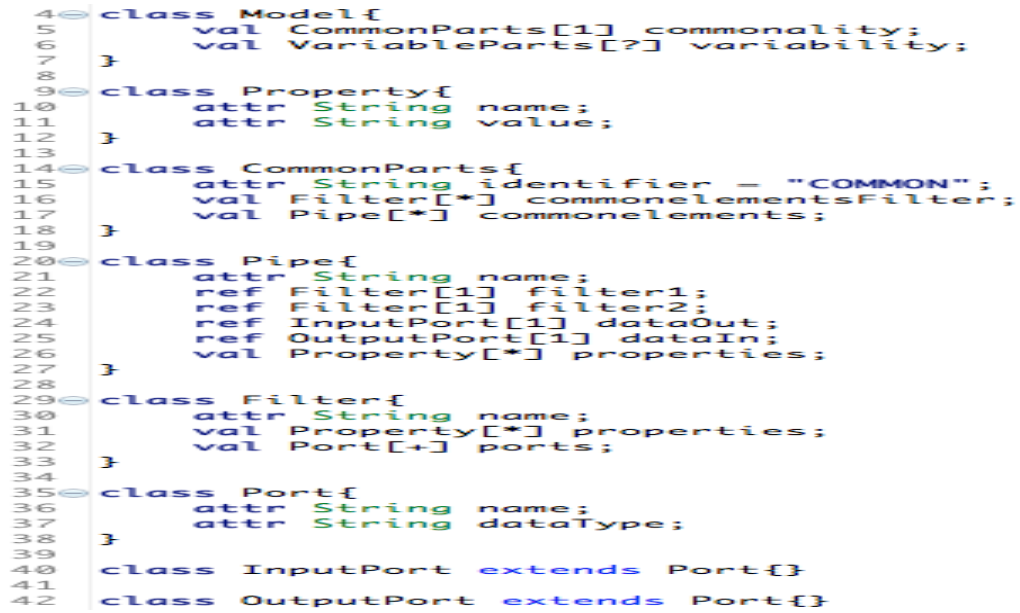

Figure 6.18: Reference architecture pipe-and-filter viewpoint metamodel definition

6.1.6 Pipe-and-Filter Viewpoint

A snippet of Emfatic code for metamodel definition of reference architecture pipe-and-filter viewpoint can be seen in Figure 6.18. The lines between 20-27 and 29-33 are used to declare pipe and filter metaclasses respectively. Emfatic code for the metamodel definition of application architecture pipe-and-filter viewpoint is similar to the one for reference architecture pipe-and-filter viewpoint that is demonstrated in Figure 6.18.

In Figure 6.19, a self-explanatory snippet of EVL code for validating the reference architecture pipe-and-filter viewpoint metamodel-model pair is given, which is used to check one of the constraints defined for the viewpoint.

In Figure 6.20, EGL code used to write filters, which are common elements, with their ports and properties into an XML file is demonstrated.

In the code for finding the status of attachment relation, the main idea is similar to one developed for uses relations that is partly depicted in Figure 6.8

```
context Pipe{
    // Pipe-and-Filter viewpoint constraint is handled here
    // Linked filters must be on the same page regarding the type of data being transferred
    constraint DataSame{
        check : self.dataOut.dataType = self.dataIn.dataType
        message : "The type of data being transferred must be same: " + self.dataOut.name + " - "
         + self.dataOut.dataType + " and " + self.dataIn.name + " - " + self.dataIn.dataType
    }
    // The connected filters cannot be same
    constraint FiltersSame{
        check : self.filter1 <> self.filter2
        message : "The pipe " + self + " cannot connect same filters: "
         + self.filter1.name + " - " + self.filter2.name
    }
}
```

Figure 6.19: Validation code for reference architecture pipe-and-filter viewpoint

```
<commonality>
  [%if (CommonParts.all().commonelementsFilter.name.isDefined()){
      for (names in CommonParts.all().commonelementsFilter){
          for (namesC in names) {%]
  <commonfilter name="[%=namesC.name%]"/>
  [%if (namesC.ports.isDefined()){
          for(pro in namesC.ports){%]
    <commonport elementtype="[%=pro.type().name%]" name="[%=pro.name%]" dataType="[%=pro.dataType%]"/>
          [%}%]
      [%}%]
  [%if (namesC.properties.isDefined()){
          for(pro in namesC.properties){%]
    <commonfilterproperties name="[%=pro.name%]" value="[%=pro.value%]"/>
          [%}%]
      [%}%]
  [%}}%]
```

Figure 6.20: M2T transformation code for reference architecture pipe-and-filter viewpoint

and, lastly, the generated reflexion models can be found in Section 6.2.6.

6.2 Case Study

In order to test our approach we created a case study that has reference architecture and two different derived application architectures. In this section, considering viewpoint type, we provide the OVM of the case study, the models with their definition process (only for the reference architecture model, the ones for the application architecture models are similar to it) and result of the architecture conformance analysis of the case. In addition, since all the generated XML based reflexion models are similar, we only provide the one for decomposition viewpoint type.

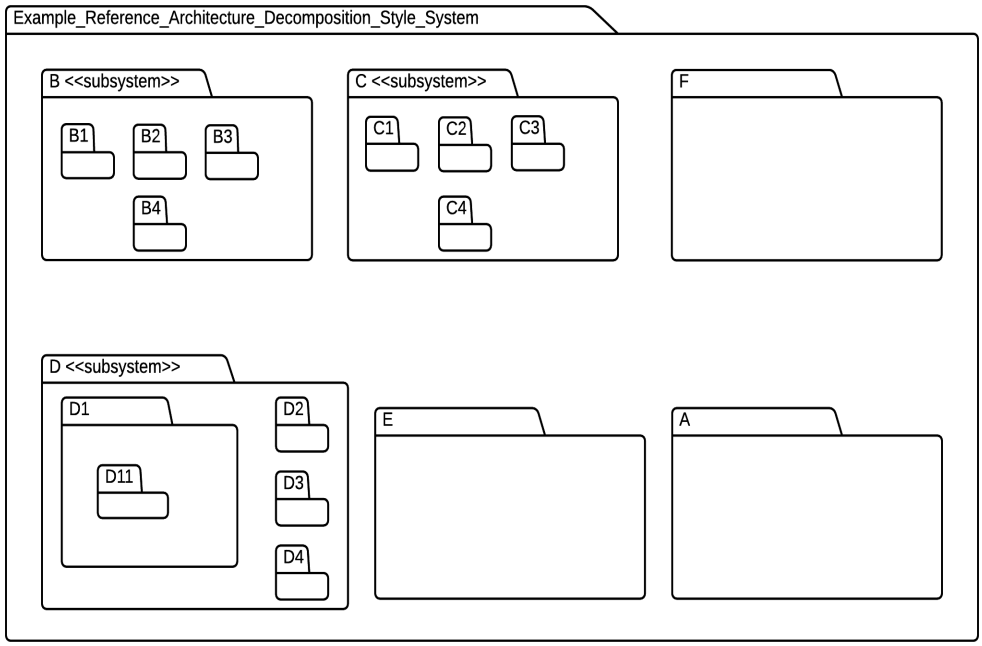

Figure 6.21: Reference architecture decomposition viewpoint of the case study

6.2.1 Decomposition Viewpoint

The reference architecture decomposition viewpoint of the case, which can be seen in Figure 6.21, has 3 subsystems and 16 modules. The details are as follows:

- Subsystem B includes 4 modules that are B1, B2, B3 and B4.
- Subsystem C includes 4 modules that are C1, C2, C3 and C4.
- Subsystem D includes 4 modules that are D1, D2, D3 and D4.
- Module D1 includes 1 module that is D11.
- Module A, E and F are not included in any element.

The OVM of the case study for the decomposition viewpoint is given in Figure 6.22, which has 5 variation points, 11 variants and 7 variability constraints. The details are as follows:

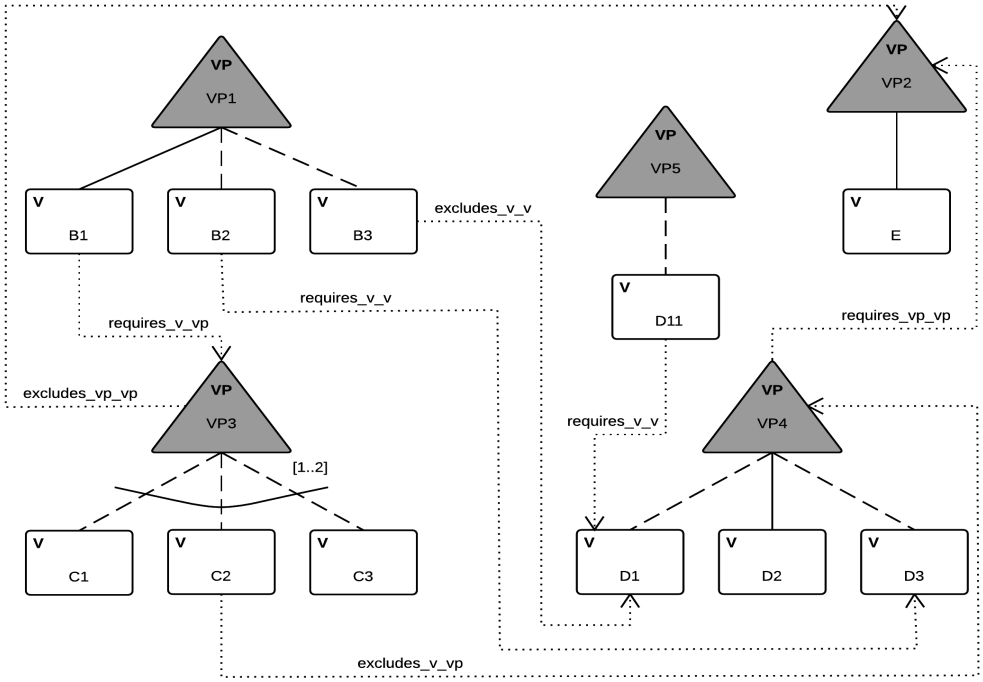

Figure 6.22: OVM of the case study for decomposition viewpoint

- Variation point VP1 has one mandatory (solid line) variant B1 and two optional (dashed line) variants B2 and B3.
- Variation point VP2 has one mandatory variant E.
- Variation point VP3 has 3 alternative (dashed line with a twisted line on) variants C1, C2 and C3, and states that at least 1 and at most 2 of the variants can be selected.
- Variation point VP4 has 2 optional variants D1 and D3, and 1 mandatory variant D2.
- Variation point VP5 has 1 optional variant D11.
- Selection of variant B1 requires realization of variation point VP3.
- Selection of variant B2 requires selection of variant D3.
- Selection of variant B3 excludes selection of variant D1.

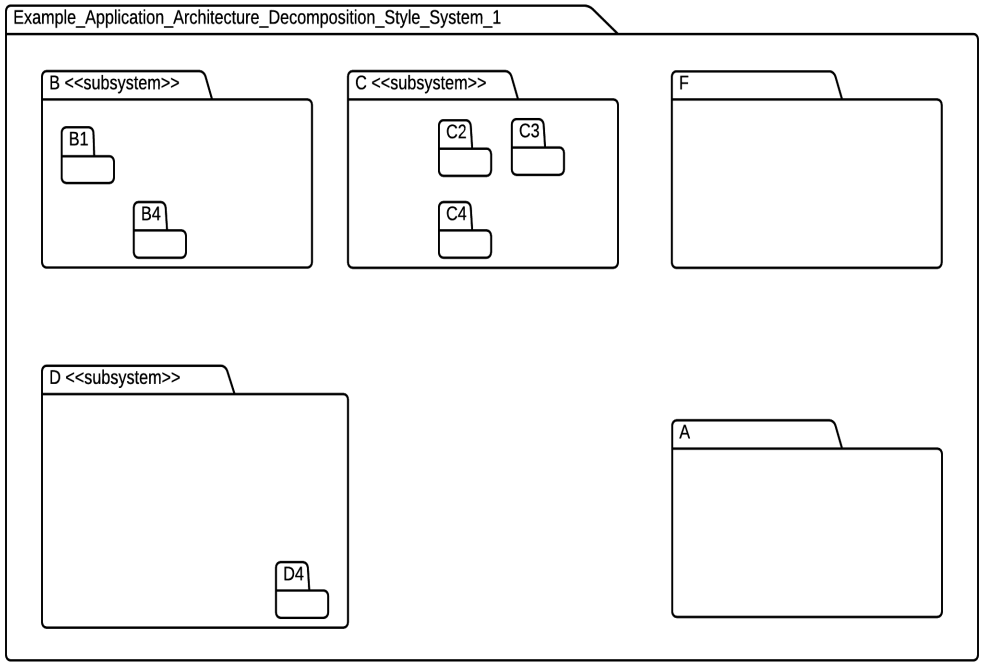

Figure 6.23: First application architecture decomposition viewpoint

- Realization of variation point VP3 excludes realization of variation point VP2.
- Selection of variant C2 excludes realization of variant point VP4.
- Selection of variant D11 requires selection of variant D1.
- Realization of variation point VP4 requires realization of variation point VP2.

By examining Figure 6.21 and 6.22, it can be stated that the common elements of the case study for decomposition viewpoint are module A, F, B4, C4 and D4, and subsystem B, C and D.

The two derived application architecture decomposition viewpoints of the case are demonstrated in Figure 6.23 and 6.24. The OVM of the case study for decomposition viewpoint was considered when the two application architecture decomposition viewpoints were derived.

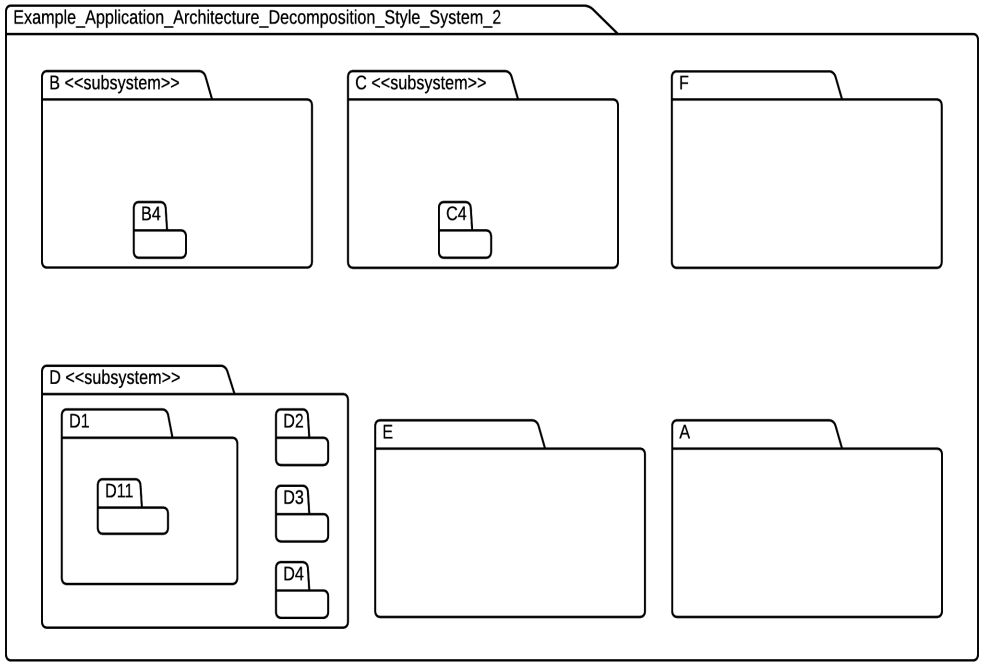

Figure 6.24: Second application architecture decomposition viewpoint

In order to test our approach using the first application architecture model, we injected following faults to the model:

- Variant module C1 is added into the model, which causes the violation of the alternative relation's max attribute. The alternative relation variant C1 belongs to allows selecting at most 2 variants but the number of selected variants is now 3.
- Variant module E is added into the model, which causes the violation of the variability constraint "VP3_E_VP2".
- Common elements module A and F are removed from the model.

The expected output for the test using the first application architecture model is that all variants, which are module C1, C2 and C3, of the variation point VP3 are marked as Variant Absence, the variant E is marked as Variant Absence, the

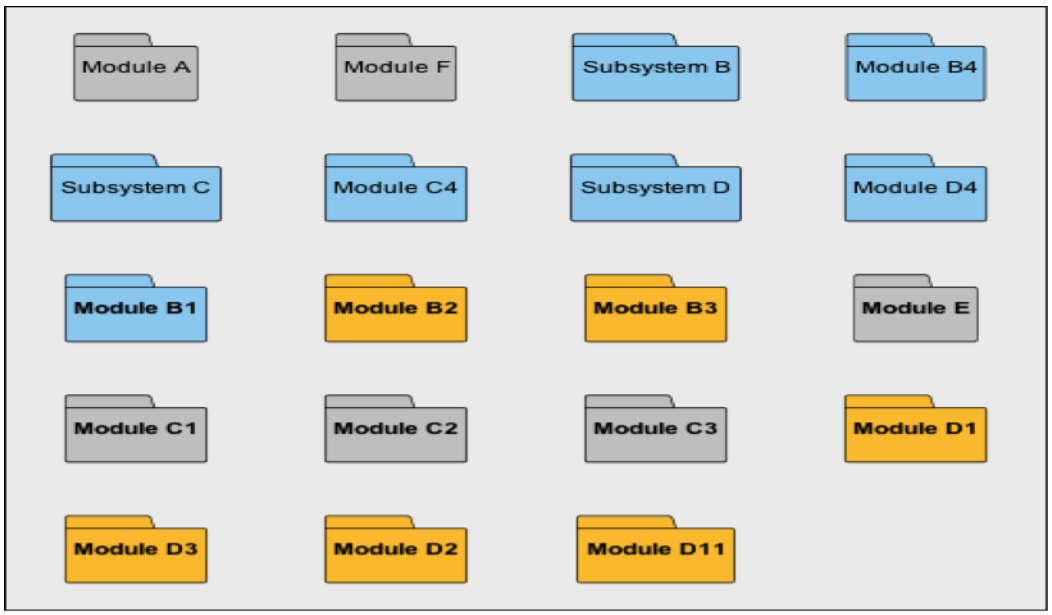

Figure 6.25: Generated image based reflexion model for the case study's decomposition viewpoint model - 1

common elements module A and F are marked as Common Absence, the common elements subsystem B, C, D and module B4, C4 and D4 are marked as Common Convergence, the variant B1 selected in the application architecture is marked as Variant Convergence, and the variants B2, B3, D1, D11, D2 and D3 are marked as not selected variants. The actual output of the test, which is generated by our tool, as an image based reflexion model can be seen in Figure 6.25 and as a textual based, i.e. XML file, reflexion model can be seen in Appendix-G.

In Figure 6.25 and 6.26, the convergence elements are given in color blue, the absence elements are given in color gray, the delta elements are given in color red and the not selected variants are given in color yellow. In addition, the variant elements have their name in bold font, whereas the common elements have their name in standard font.

The injected faults to the second application architecture model for testing are as follows:

- Common elements module A and subsystem B are removed from the model.

- A delta element module X is added into the model.
- Variant module D1 is removed from the model, which causes the violation of the variability constraint "D11_R_D1".
- Variant module D2 is removed from the model, which is a mandatory variant in the variation point VP4.
- Realization of the variation point VP2 is removed, i.e. the variant E is removed, from the model, which then causes the violation of the variability constraint "VP4_R_VP2".

The expected output for the test using the second application architecture model is that the module A and B4 (since B4's parent element (subsystem B) is now missing, which is not matched with the form of B4 given in the reference architecture), and subsystem B are marked as Common Absence, the module X is marked as Delta, the variant module D1 is marked as Variant Absence, the variant module D11 is marked as Variant Absence since its parent element (variant D1) is missing in the application architecture model, the variant module D2 is marked as Variant Absence since it is a mandatory element in the realized variation point VP4, the variant module E is marked as Variant Absence, the modules C4, D4 and F, and subsystems C and D are marked as Common Convergence, the variant D3 is marked as Variant Convergence, and the variant modules C1, C2, C3, B1, B2 and B3 are marked as not selected variants. The actual output of the test can be seen in Figure 6.26.

6.2.2 Uses Viewpoint

Since the uses viewpoint definition depends on the decomposition viewpoint definition, our case's uses viewpoint models and corresponding OVM are linked to the decomposition viewpoint models and OVM given in Section 6.2.1. The reference architecture uses viewpoint of the case, which can be seen in Figure 6.27, has 22 uses relations such as module D2 uses module E, subsystem C uses module F, etc.

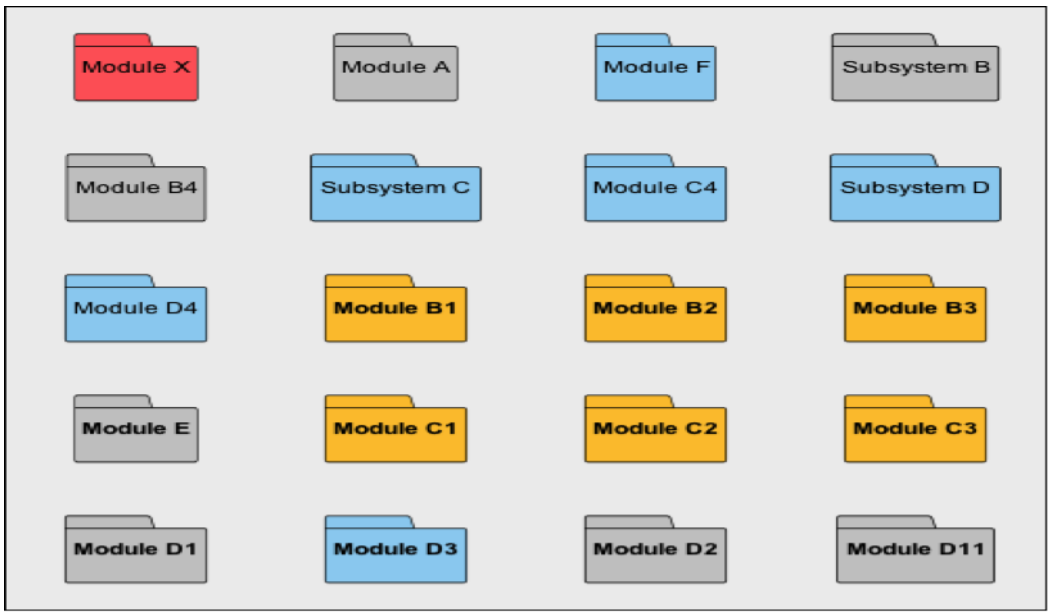

Figure 6.26: Generated image based reflexion model for the case study's decomposition viewpoint model - 2

The two derived application architecture uses viewpoints of the case are demonstrated in Figure 6.28 and 6.29. The OVM of the case study for decomposition viewpoint, which is demonstrated in Figure 6.22, was considered when the two application architecture uses viewpoints were derived.

In order to test our approach using the first application architecture model, we injected following faults to the model:

- Delta module P is added into the model and new uses relations such as P uses A and A uses P are established.
- Variant element module C2 is removed from the model.
- Common element module F is removed from the model.
- Uses relation A uses C4 is removed from the model.
- Uses relation D4 uses B4 is altered to D4 uses C4 in the model.

The expected output for the test using the first application architecture model

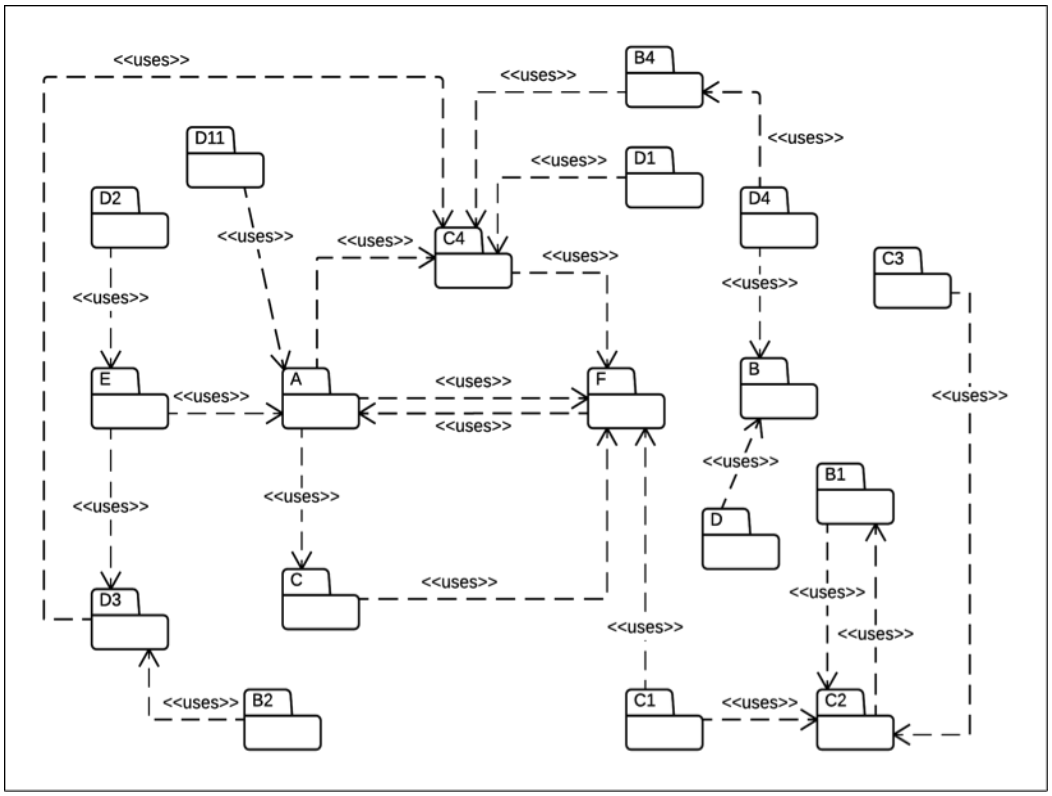

Figure 6.27: Reference architecture uses viewpoint of the case study

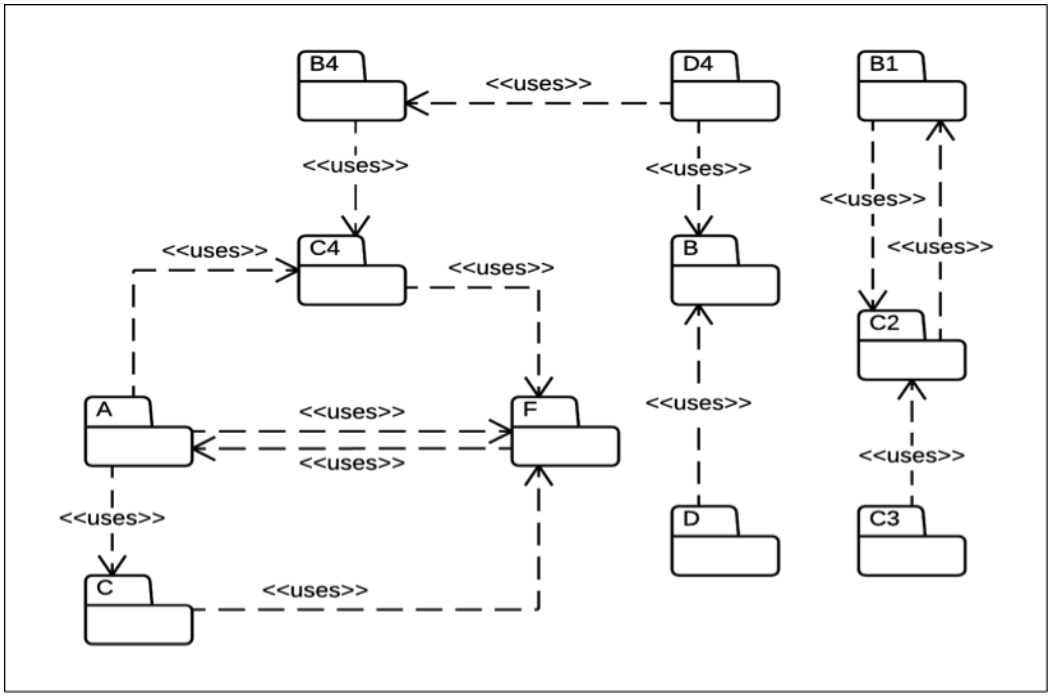

Figure 6.28: First application architecture uses viewpoint

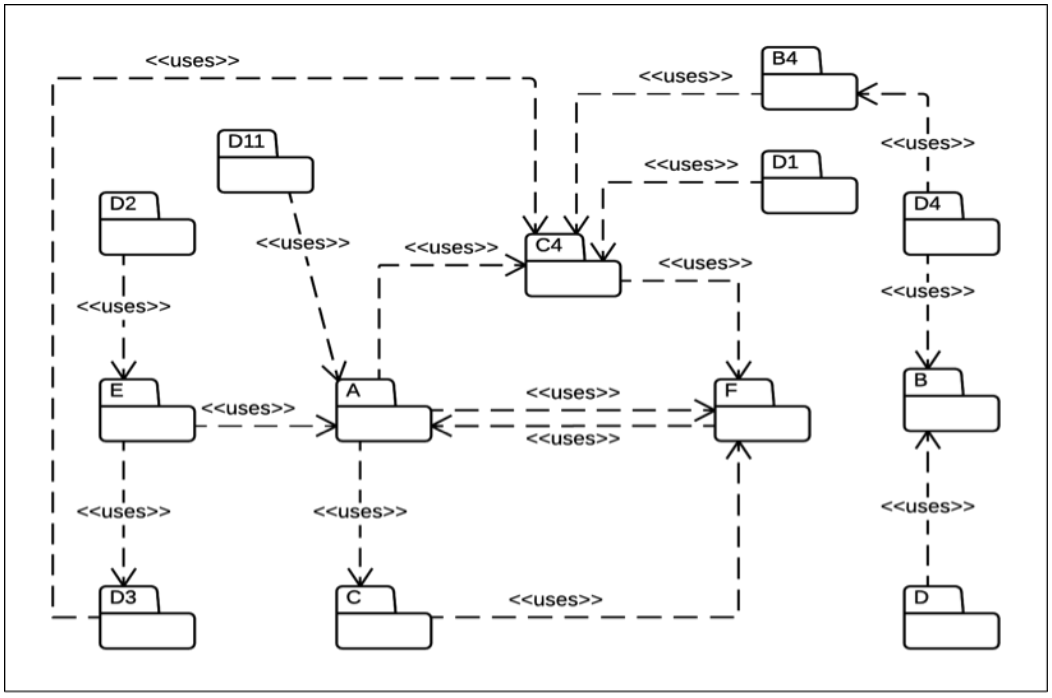

Figure 6.29: Second application architecture uses viewpoint

is that the uses relations P uses A and A uses P are marked as Delta, the uses relations B1 uses C2, C3 uses C2, C4 uses F, C uses F, A uses F, F uses A, A uses C4 and D4 uses B4 are marked as Absence, the uses relation D4 uses C4 is marked as Delta, the uses relations D4 uses B, A uses C, B4 uses C4 and D uses B are marked as Convergence, and the uses relations D2 uses E, E uses D3, B2 uses D3, D11 uses A, D1 uses C4, C2 uses B1, C1 uses F and C1 uses C2 are marked as Not Selected Uses Relation. The actual output of the test, which is generated by our tool, as an image based reflexion model can be seen in Figure 6.30. This reflexion model is based on Design Structure Matrix (DSM) idea [56] that is an alternative concise representation for architecture viewpoints introduced in [4]. In a DSM for uses viewpoint, the column represent source/using element and the row represent target/used element of a uses relation. In addition, if there is a uses relation between ith column element and jth row element, i.e. element i uses element j, then the corresponding cell is filled with "1". The color of "1" indicates the status of the uses relation such as Convergence (Blue), Absence (Cyan), Delta (Red) and Not Selected Uses Relation (Yellow). In addition, same color pattern is applied to source/target elements given in the columns/rows to indicate their

status and also common elements are given in standard font and variant elements are given in bold font, e.g. common convergence elements are typed in color blue and standard font, variant absence elements are typed in color cyan and bold font, not selected variants are typed in color yellow and bold font, etc.

In order to test our approach using the second application architecture model, we injected following faults to the model:

- Variant element module D3 is removed from the model.
- Uses relation E uses A is removed from the model.
- Uses relation D1 uses C4 is altered to D1 uses F in the model.
- Delta module P is added into the model and new uses relation P uses B4 is established.

The expected output for the test using the second application architecture model is that uses relations E uses D3, E uses A and D1 uses C4 are marked as Absence, uses relations D3 uses C4, B2 uses D3, C1 uses F, C1 uses C2, C2 uses B1, B1 uses C2 and C3 uses C2 are marked as Not Selected Uses Relation, uses relations D1 uses F and P uses B4 are marked as Delta, and uses relations D11 uses A, D2 uses E, A uses C, C uses F, A uses F, F uses A, A uses C4, C4 uses F, B4 uses C4, D4 uses B4, D4 uses B and D uses B are marked as Convergence. The actual output of the test can be seen in Figure 6.31.

6.2.3 Generalization Viewpoint

The case study's reference architecture generalization viewpoint, which is demonstrated in Figure 6.32 and 6.33, introduces the modules (i.e. classes) and interfaces included in the model elements of the decomposition viewpoint model given in Section 6.2.1. In addition, OVM for the generalization viewpoint of the case is same as the one defined for the decomposition viewpoint that is demonstrated in

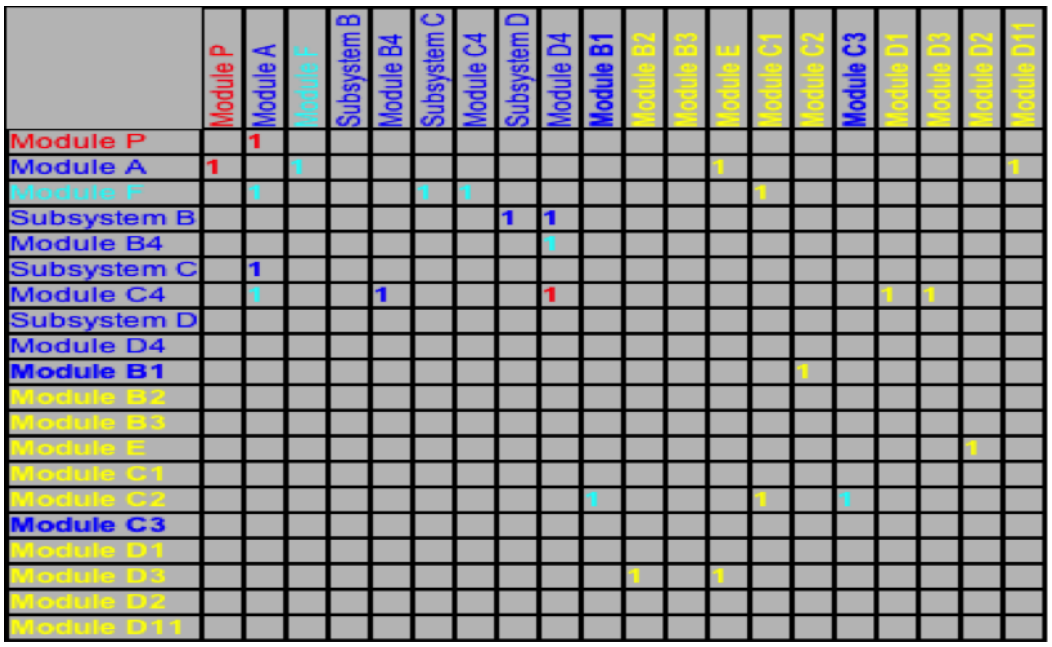

Figure 6.30: Generated image based reflexion model for the case study's uses viewpoint model - 1

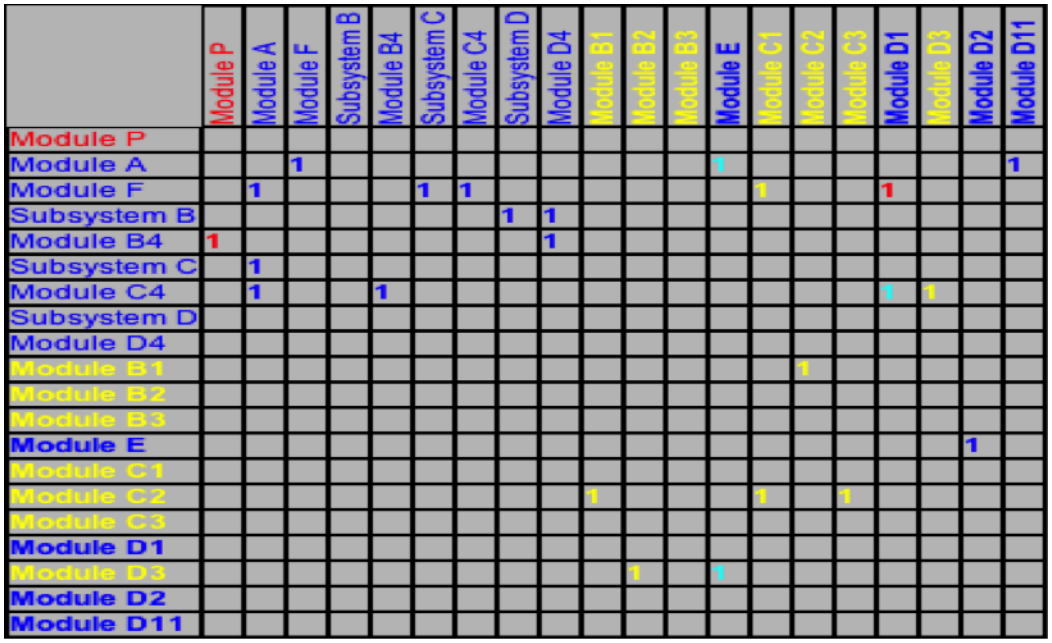

Figure 6.31: Generated image based reflexion model for the case study's uses viewpoint model - 2

Figure 6.22. The modules and interfaces are affected by the variability constraints as the way their super-model element is affected. For example, the model element variant module B1 includes two interfaces called B1_1 and B1_2, and three modules called B1_3, B1_4 and B1_5. Those five sub-model elements have the same variability constraints as B has.

In Figure 6.32 and 6.33 the dashed lines with an arrowhead indicates the Interface Implementation relation and the solid lines with an arrowhead indicates either the Class Inheritance relation or Interface Inheritance relation. Likewise, the interfaces are highlighted with keyword "<<interface>>" and the classes are given as a simple class - UML notation.

The Class Inheritance relation can be established only between one parent class and one child class. For example, in Figure 6.32, the module/class A_2 (child) is-a module/class F_2 (parent). The Interface Inheritance relation can be established only between one parent interface and one child interface. For instance, the interface D1_1 (child) is-a interface C4_3 (parent) in Figure 6.32. The Interface Implementation relation can be established only between one parent interface and one child class. For example, in Figure 6.32, the class B1_3 (child) is-a interface B1_2 (parent).

Since the number of variant elements for the variation point VP3 that includes variants C1, C2 and C3 (see Figure 6.22) is increased with introduction of new modules/interfaces to the case study, we changed the value of "max" attribute of the Alternative relation of that variation point as "10" which was previously "2". Therefore, the constraint remains consistent with the number of variant elements listed in the variation point VP3.

The two derived application architecture generalization viewpoints of the case are demonstrated in Figure 6.34 and 6.35. In order to test our approach using the first application architecture model, we injected following faults to the model:

- Delta module P is added into the model and new Class Inheritance relations such as P is-a A_1 and A_3 is-a P are established.

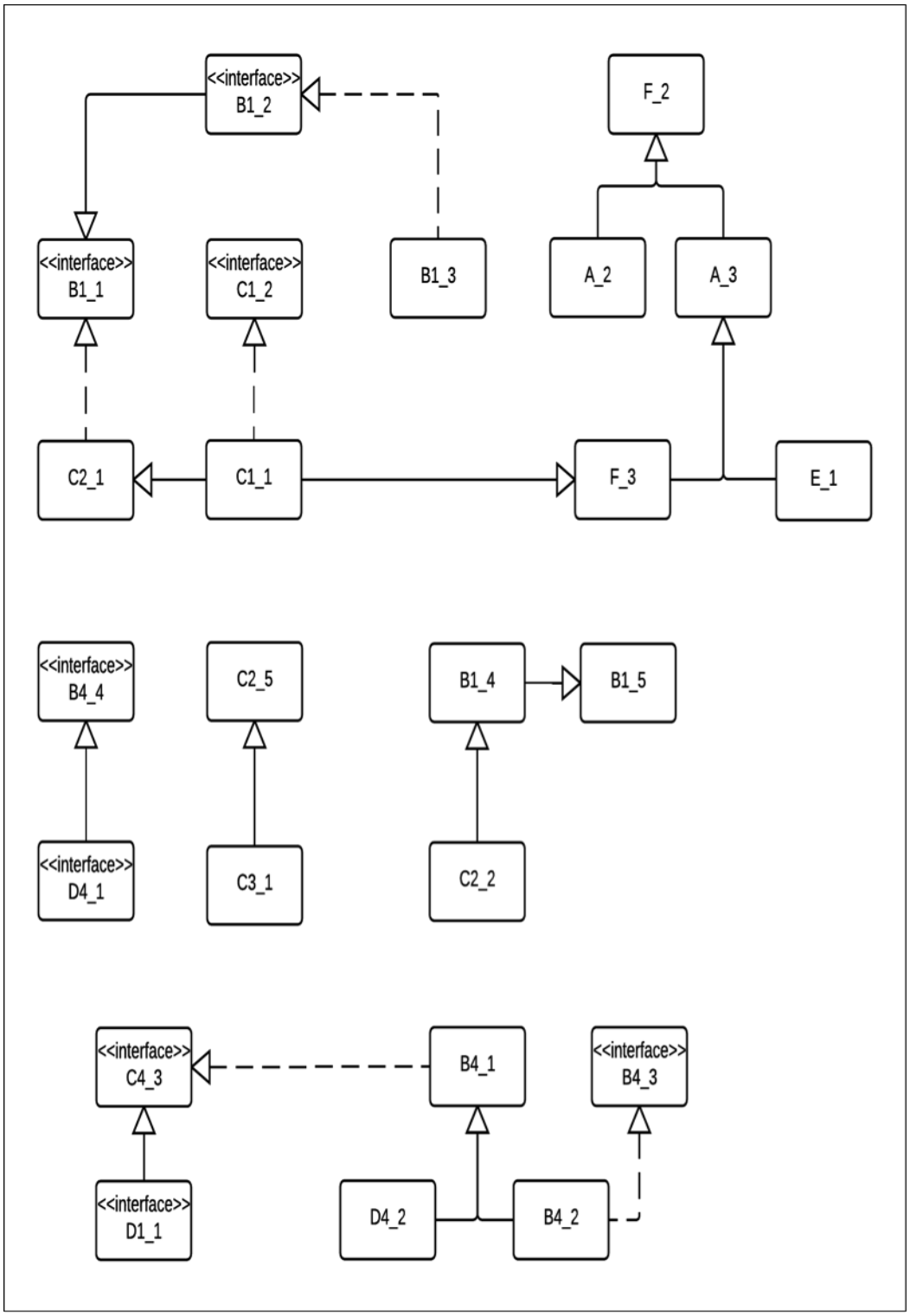

Figure 6.32: Reference architecture generalization viewpoint of the case study - 1

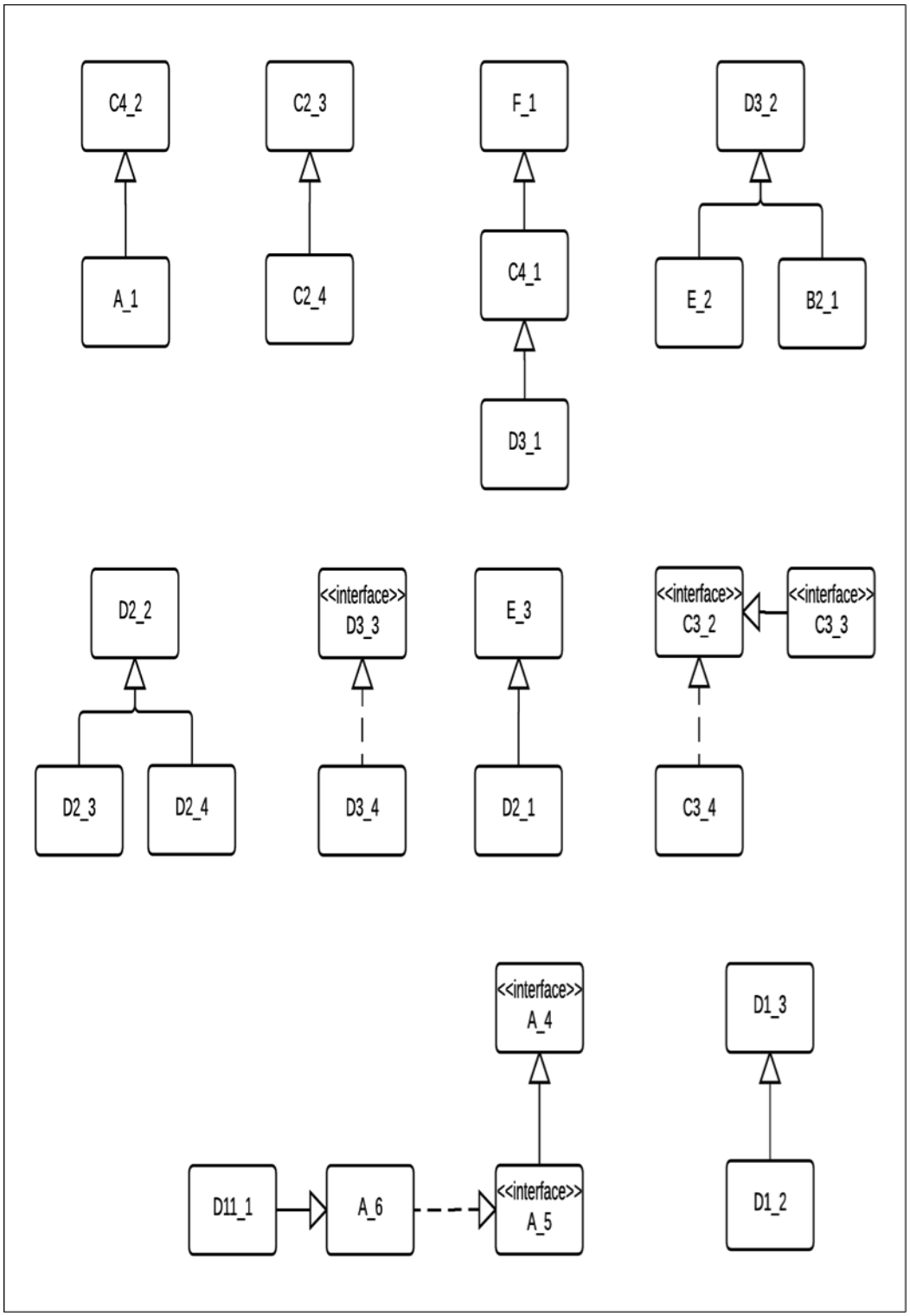

Figure 6.33: Reference architecture generalization viewpoint of the case study - 2

- Variant element module C2_5 is removed from the model.
- Common element module F_2 is removed from the model.
- Interface Inheritance relation D4_1 is-a B4_4 is removed from the model.
- Class Inheritance relation D4_2 is-a B4_1 is altered to D4_2 is-a C4_2 in the model.
- Interface Implementation relation A_6 is-a A_5 is removed from the model.
- Variant element module C2_4 is removed from the model.

The expected output for the test using the first application architecture model is that Class Inheritance relations P is-a A_1, A_3 is-a P and D4_2 is-a C4_2 are marked as Delta, Class Inheritance relations C3_1 is-a C2_5, A_2 is-a F_2, A_3 is-a F_2 and D4_2 is-a B4_1 are marked as Absence, Interface Inheritance relation D4_1 is-a B4_4 is marked as Absence, Interface Implementation relation A_6 is-a A_5 is marked as Absence, Class Inheritance relations C2_4 is-a C2_3, E_1 is-a A_3, D11_1 is-a A_6, C1_1 is-a F_3, D3_1 is-a C4_1, D2_1 is-a E_3, C1_1 is-a C2_1, D1_2 is-a D1_3, B2_1 is-a D3_2, E_2 is-a D3_2, D2_3 is-a D2_2 and D2_4 is-a D2_2 are marked as Not Selected Generalization Relation, Class Inheritance relations F_3 is-a A_3, C4_1 is-a F_1, B4_2 is-a B4_1, A_1 is-a C4_2, B4_1 is-a C4_3, C2_2 is-a B1_4 and B1_4 is-a B1_5 are marked as Convergence, Interface Inheritance relations A_5 is-a A_4, B1_2 is-a B1_1 and C3_3 is-a C3_2 are marked as Convergence, Interface Implementation relations B4_2 is-a B4_3, C2_1 is-a B1_1, B1_3 is-a B1_2 and C3_4 is-a C3_2 are marked as Convergence, Interface Inheritance relation D1_1 is-a C4_3 is marked as Not Selected Generalization Relation, and Interface Implementation relations C1_1 is-a C1_2 and D3_4 is-a D3_3 are marked as Not Selected Generalization Relation. The actual output of the test, which is generated by our tool, as an image based reflexion model can be seen in Figure 6.36. The image based reflexion model is in form of the DSM whose structure to demonstrate the status of relations remains same as explained in Section 6.2.2. However, for the generalization viewpoint, ith column element represents the child element that generalizes the parent element and jth row element represents the parent element that is being generalized.

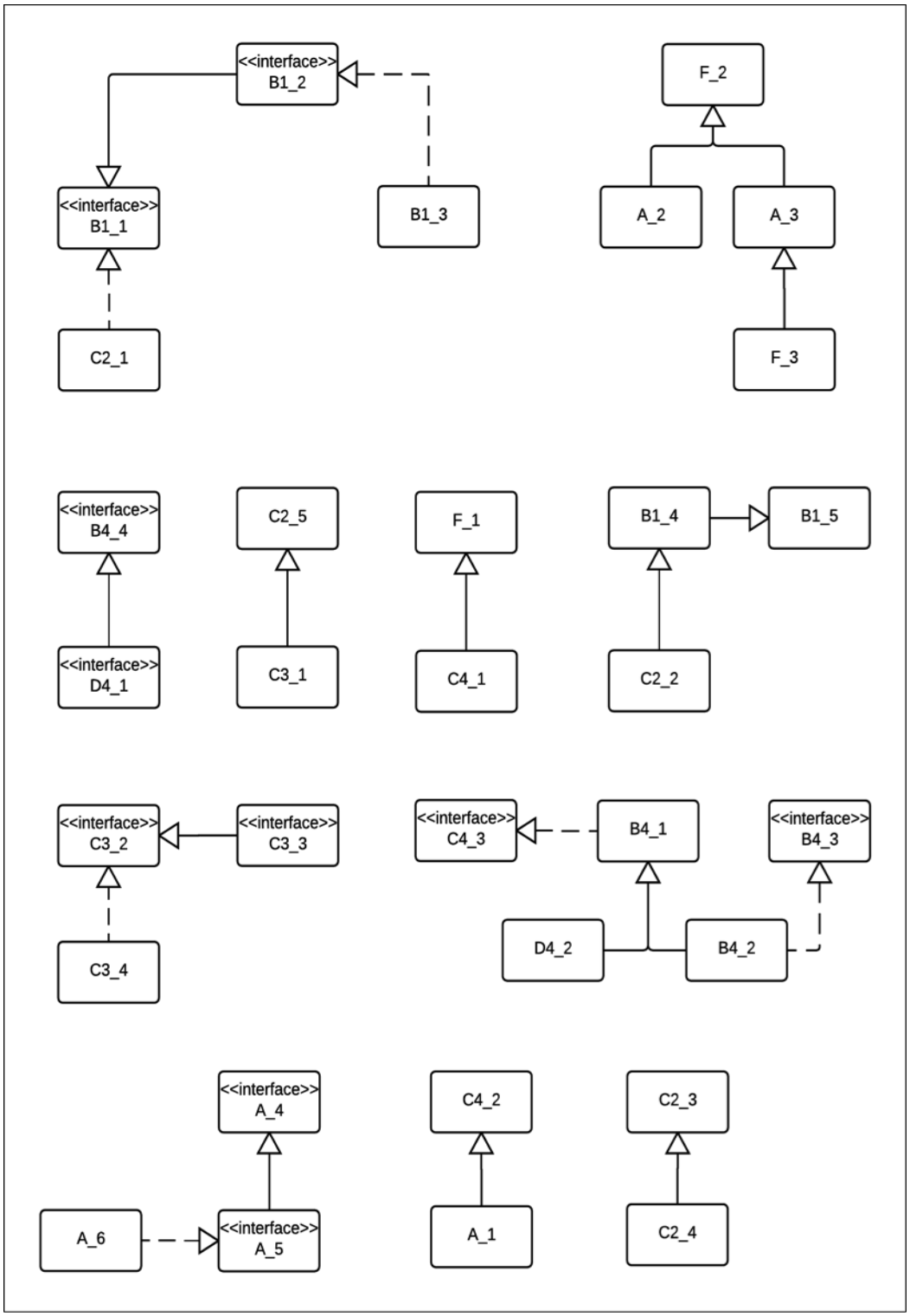

Figure 6.34: First application architecture generalization viewpoint

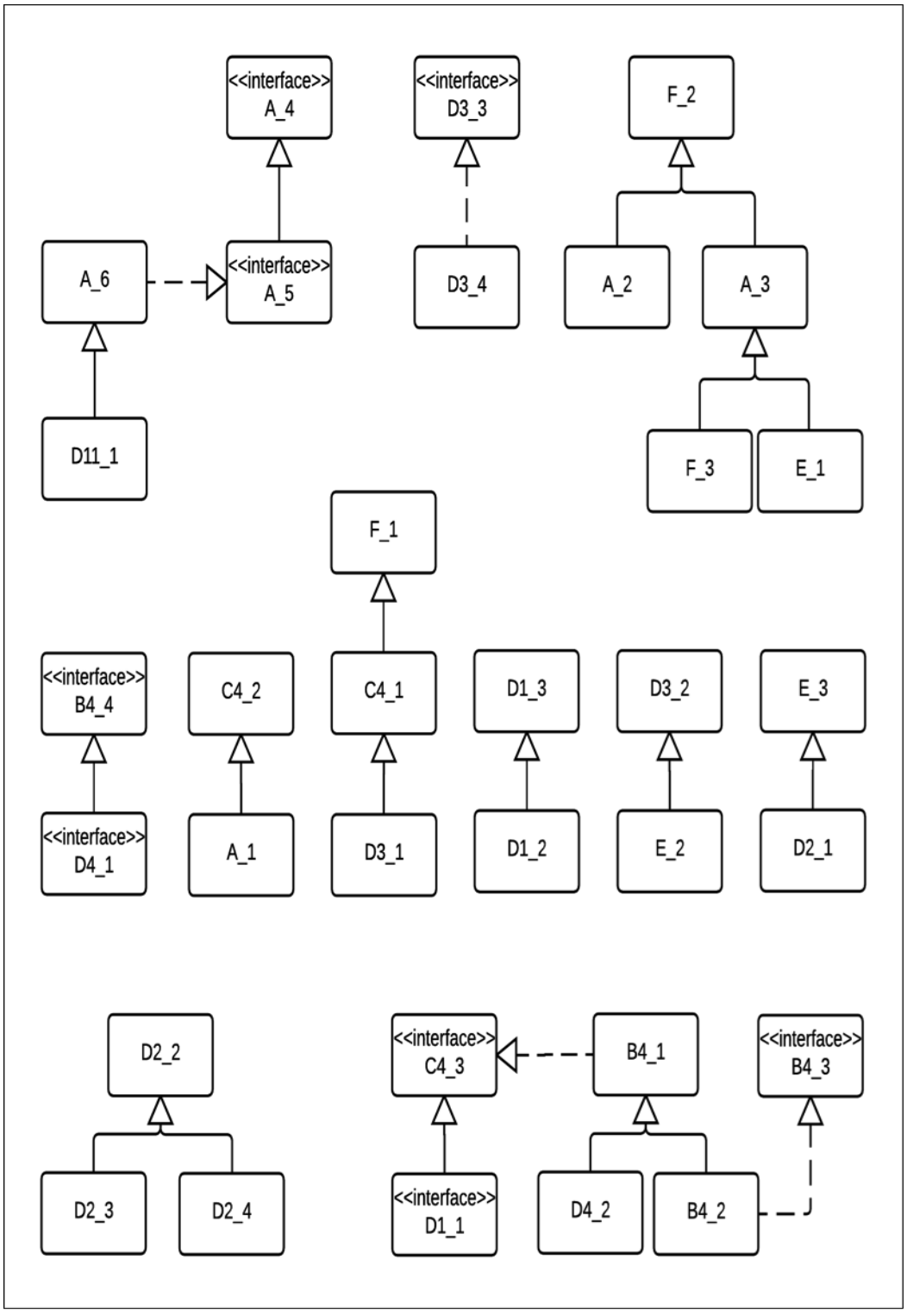

Figure 6.35: Second application architecture generalization viewpoint

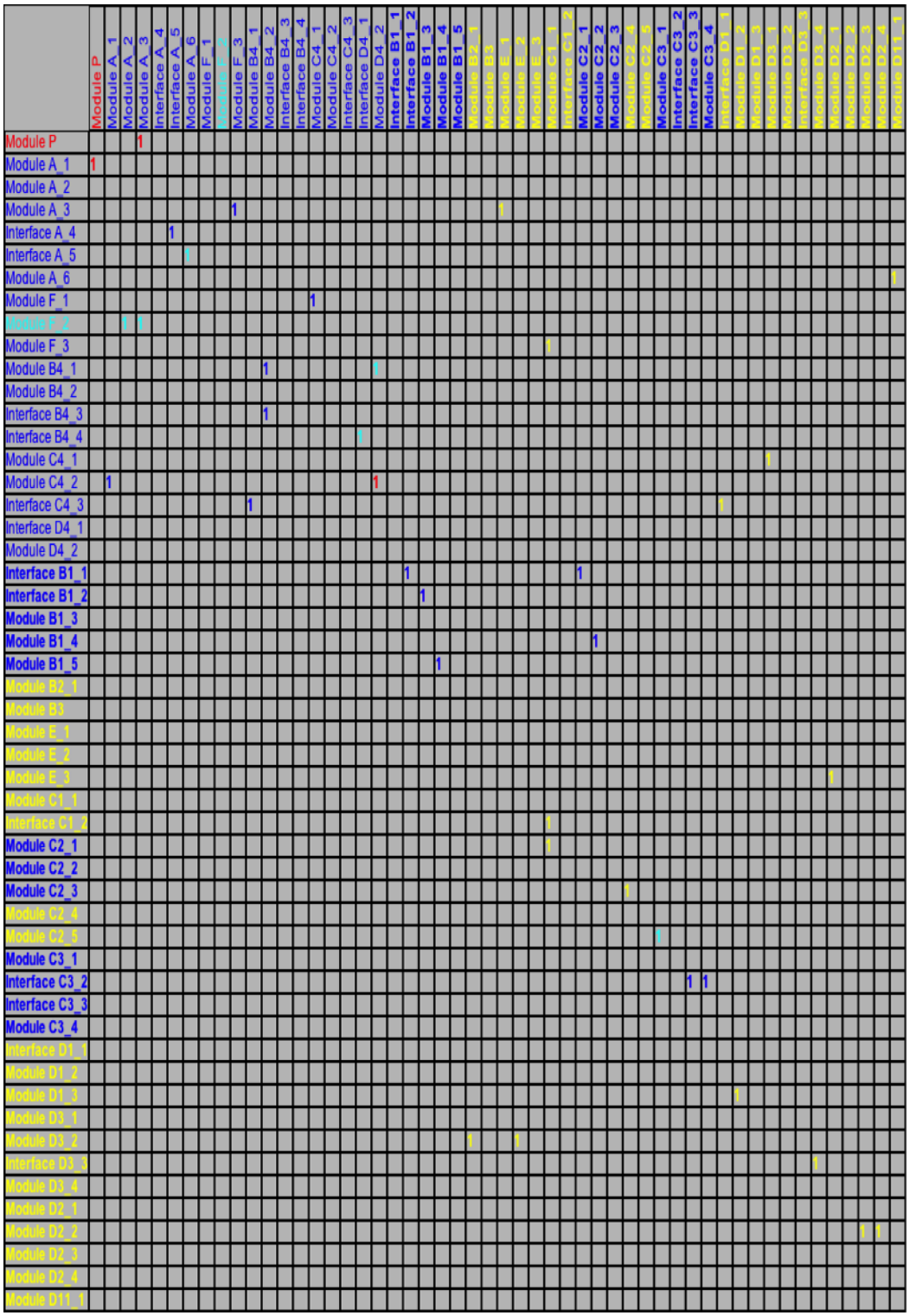

Figure 6.36: Generated image based reflexion model for the case study's generalization viewpoint model - 1

In order to test our approach using the second application architecture model, we injected following faults to the model:

- Variant element module D3_4 is removed from the model.
- Interface Implementation relation B4_2 is-a B4_3 is altered to B4_2 is-a D3_3 in the model.

The expected output for the test using the second application architecture model is that Interface Implementation relation B4_2 is-a D3_3 is marked as Delta, Interface Implementation relation B4_2 is-a B4_3 is marked as Absence, Interface Inheritance relations B1_2 is-a B1_1 and C3_3 is-a C3_2 are marked as Not Selected Generalization Relation, Interface Implementation relations C2_1 is-a B1_1, C1_1 is-a C1_2, B1_3 is-a B1_2, D3_4 is-a D3_3 and C3_4 is-a C3_2 are marked as Not Selected Generalization Relation, Class Inheritance relations C1_1 is-a C2_1, C1_1 is-a F_3, C3_1 is-a C2_5, C2_2 is-a B1_4, B1_4 is-a B1_5, C2_4 is-a C2_3 and B2_1 is-a D3_2 are marked as Not Selected Generalization Relation, Class Inheritance relations A_2 is-a F_2, A_3 is-a F_2, F_3 is-a A_3, E_1 is-a A_3, D4_2 is-a B4_1, B4_2 is-a B4_1, A_1 is-a C4_2, D3_1 is-a C4_1, C4_1 is-a F_1, E_2 is-a D3_2, D2_3 is-a D2_2, D2_4 is-a D2_2, D2_1 is-a E_3, D11_1 is-a A_6 and D1_2 is-a D1_3 are marked as Convergence, Interface Implementation relation A_6 is-a A_5 is marked as Convergence, and Interface Inheritance relations D4_1 is-a B4_4, D1_1 is-a C4_3 and A_5 is-a A_4 are marked as Convergence. The actual output of the test can be seen in Figure 6.37.

6.2.4 Layered Viewpoint

The case study's reference architecture has four main layered views that can be seen in Figure 6.38. Layer A is allowed to use all layers' services below it, i.e. it can use layer C that is allowed to use layer F. Layer C3 is allowed to use layer C2 that is allowed to use layer B1. However, in this case, layer C3 is not allowed to use layer B1. In addition, layer D4 is allowed to use B4 and layer E is allowed to

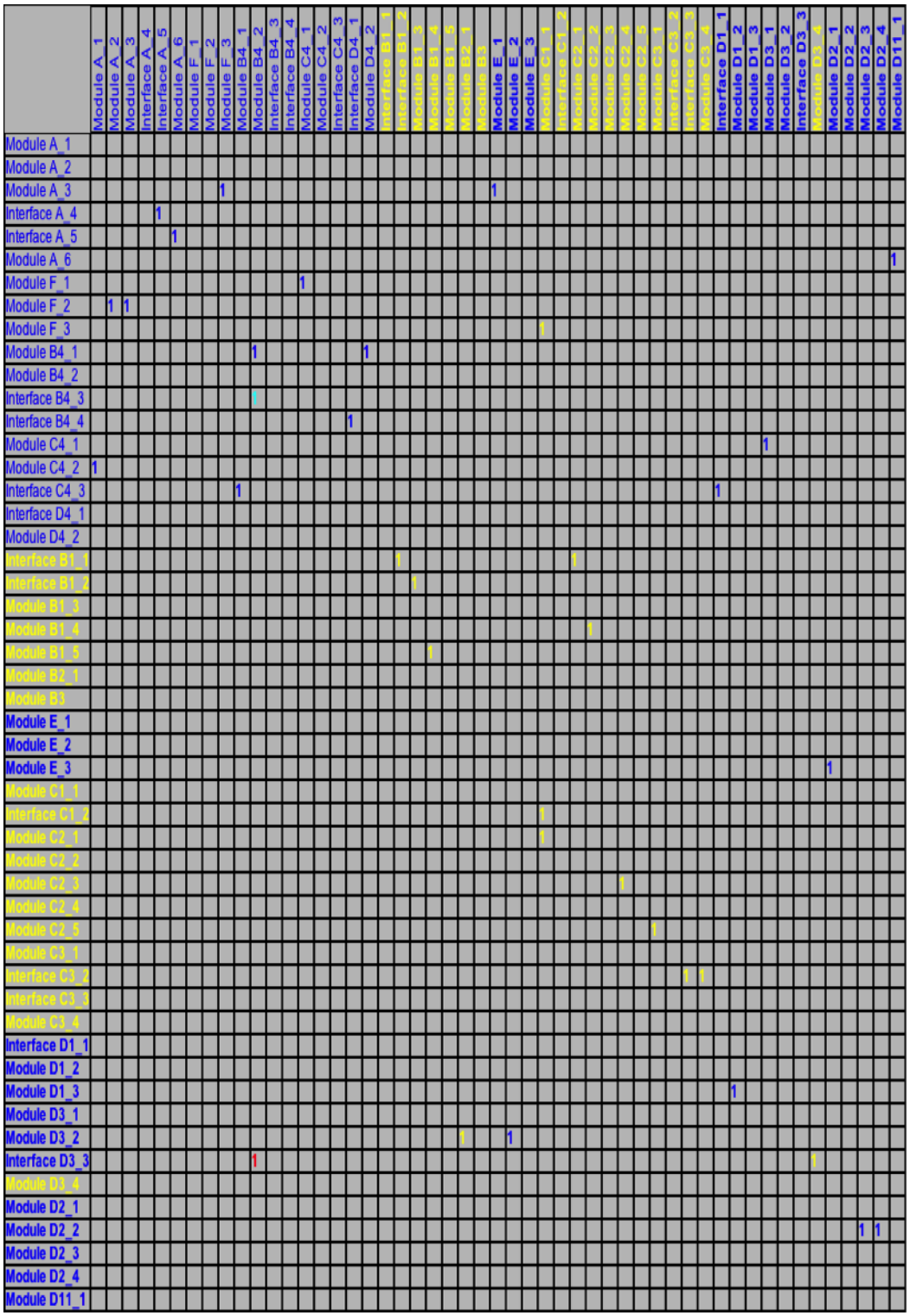

Figure 6.37: Generated image based reflexion model for the case study's generalization viewpoint model - 2

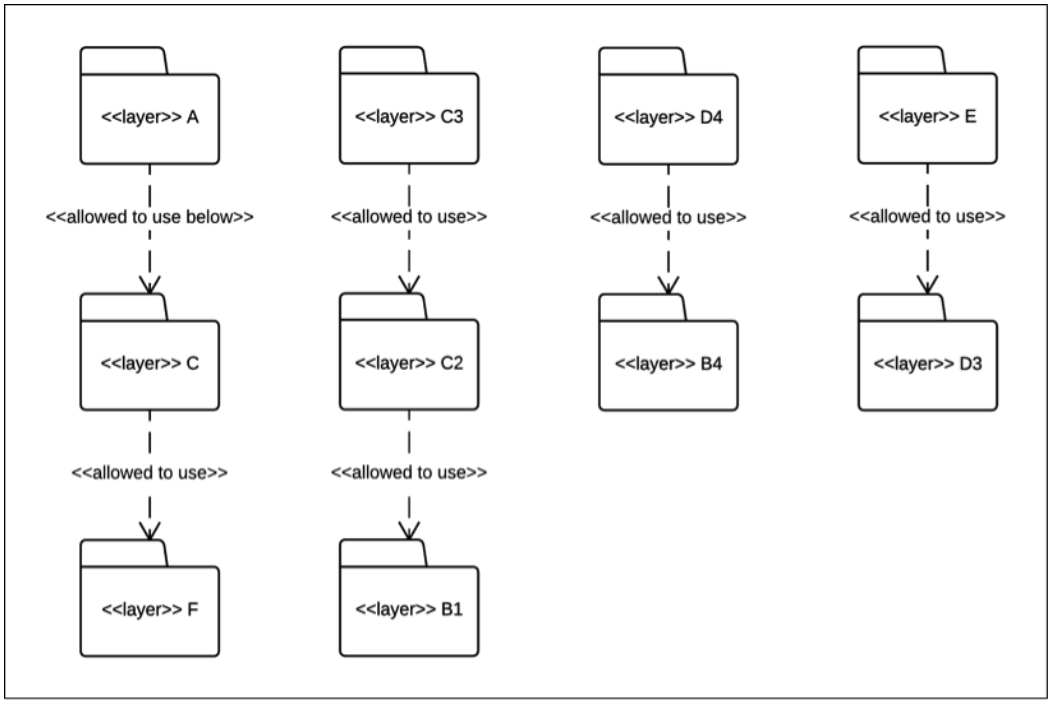

Figure 6.38: Reference architecture layered viewpoint of the case study

use layer D3. It is crucial to note that all directions are unidirectional, e.g. layer C cannot use layer A. Furthermore, the OVM, see Figure 6.22, is considered when the layered viewpoints are created with one modification that instead of module or subsystem names that are given in Section 6.2.1, the layer names are used. The reference architecture layered viewpoint has ten layers and six Allowed-to-use relations including one Allowed-to-use-below relation.

The two derived application architecture layered viewpoints of the case are demonstrated in Figure 6.39 and 6.40. In order to test our approach using the first application architecture model, we injected following faults to the model:

- Delta layer P is added into the model and new Allowed-to-use relation such as P is allowed to use D4 is established.
- Variant layer C2 is removed from the model.
- Common layer F is removed from the model.
- The relation D4 is allowed to use B4 is removed from the model.

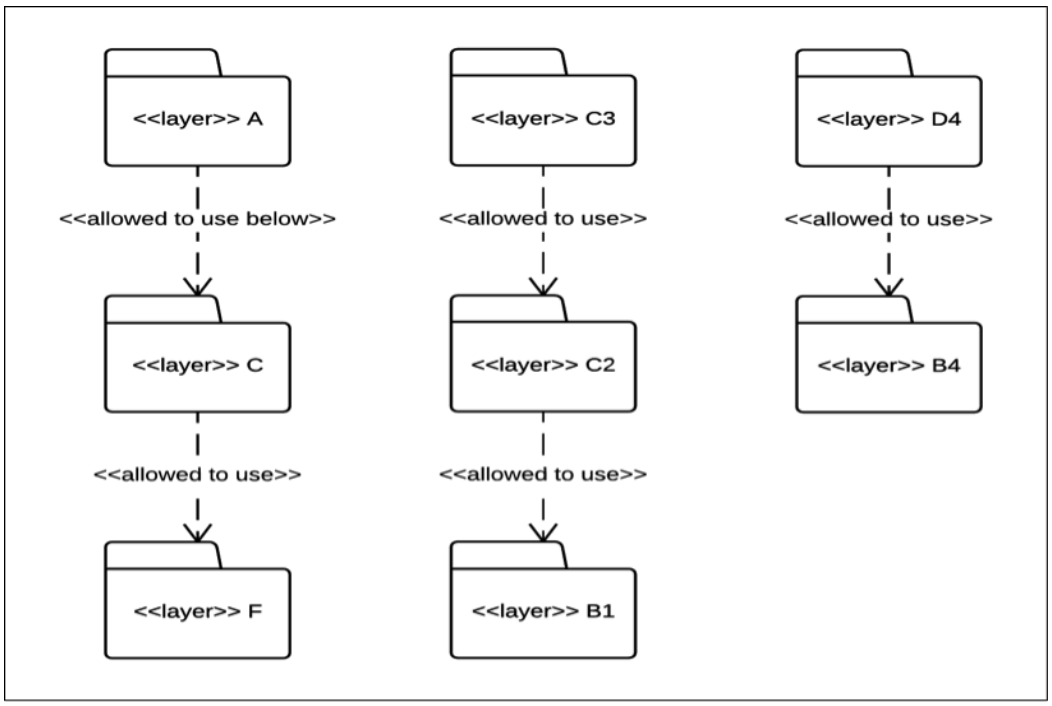

Figure 6.39: First application architecture layered viewpoint

- The relation A is allowed to use below C is altered to A is allowed to use C in the model.

The expected output for the test using the first application architecture model is that the relation P is allowed to use D4 is marked as Delta, the relations A is allowed to use below C, C is allowed to use F, D4 is allowed to use B4 and C3 is allowed to use C2 are marked as Absence, and the relations C2 is allowed to use B1 and E is allowed to use D3 are marked as Not Selected Layer Relation. The actual output of the test can be seen in Figure 6.41. The image based reflexion model is in form of the DSM whose structure to highlight the status of layer relations and layers, and the way to read column-row pairs remains same as explained in Section 6.2.2.

In order to test our approach using the second application architecture model, we injected following faults to the model:

- The relation A is allowed to use below C is altered to A is allowed to use

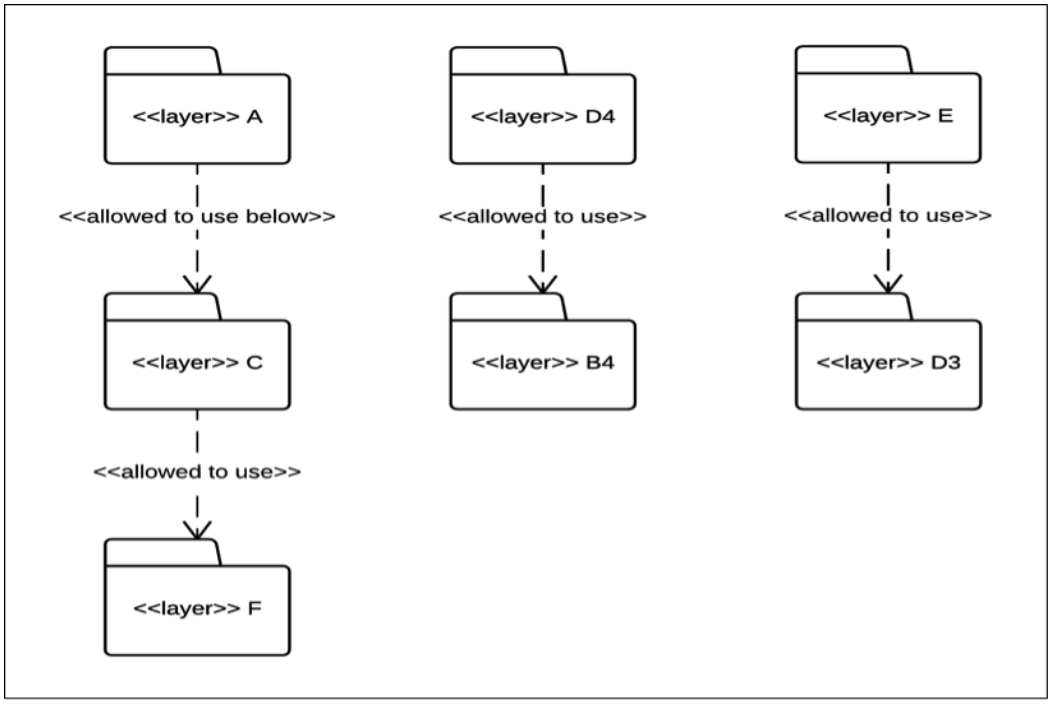

Figure 6.40: Second application architecture layered viewpoint

below B4 in the model.

- The relation C is allowed to use F is altered to C is allowed to use below F in the model.

The expected output for the test using the second application architecture model is that the relations D4 is allowed to use B4 and E is allowed to use D3 are marked as Convergence, the relation A is allowed to use below B4 is marked as Delta, the relations C is allowed to use F and A is allowed to use C are marked as Absence, and the relations C2 is allowed to use B1 and C3 is allowed to use C2 are marked as Not Selected Layer Relation. The actual output of the test can be seen in Figure 6.42.

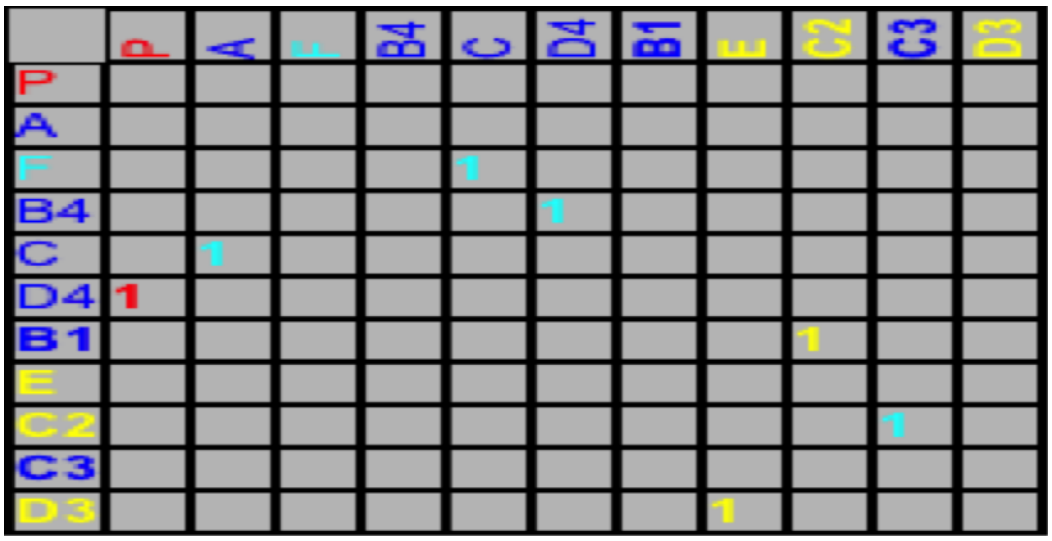

Figure 6.41: Generated image based reflexion model for the case study's layered viewpoint model - 1

Figure 6.42: Generated image based reflexion model for the case study's layered viewpoint model - 2

6.2.5 Deployment Viewpoint

The case study's reference architecture deployment viewpoint, which is given in Figure 6.43, has 4 devices that are "Device1", "Device2", "Device3" and "Device4" on which the components given in the reference architecture decomposition viewpoint (see Figure 6.21) are allocated. For the deployment viewpoint, we created a new OVM of the case, which can be seen in Figure 6.44. The meaning of the variability constraints applied in the viewpoint is same as the one applied in the decomposition viewpoint.

In the case study the hardware elements called Device2 and Device1 are common elements and the other two hardware elements called Device3 and Device4 are variant elements. In addition, the software elements called component A, F, B, B4, C, C4, D and D4 are common elements and the software elements called component B1, B2, B3, E, C1, C2, C3, D1, D11, D2 and D3 are variant elements.

The devices are connected via connection links such as "connection1", "connection3", etc. Allocated-to relations can be inferred from the Figure 6.43 such as the component B is allocated-to device Device1, the component D3 is allocated to device D4, etc. However, any type of migration relations is implicit in the same figure; hence, we should list them here explicitly:

- The execution of all components on device Device3 migrates to device Device2.
- The copy of component B1 on device Device1 migrates to device Device2.
- The copy of component B2 on device Device1 migrates to device Device2.
- The copy of component D1 on device Device4 migrates to device Device2.
- The copy of component D11 on device Device4 migrates to device Device2.
- The copy of component D4 on device Device2 migrates to device Device1.
- The component B3 on device Device1 migrates to device Device2.

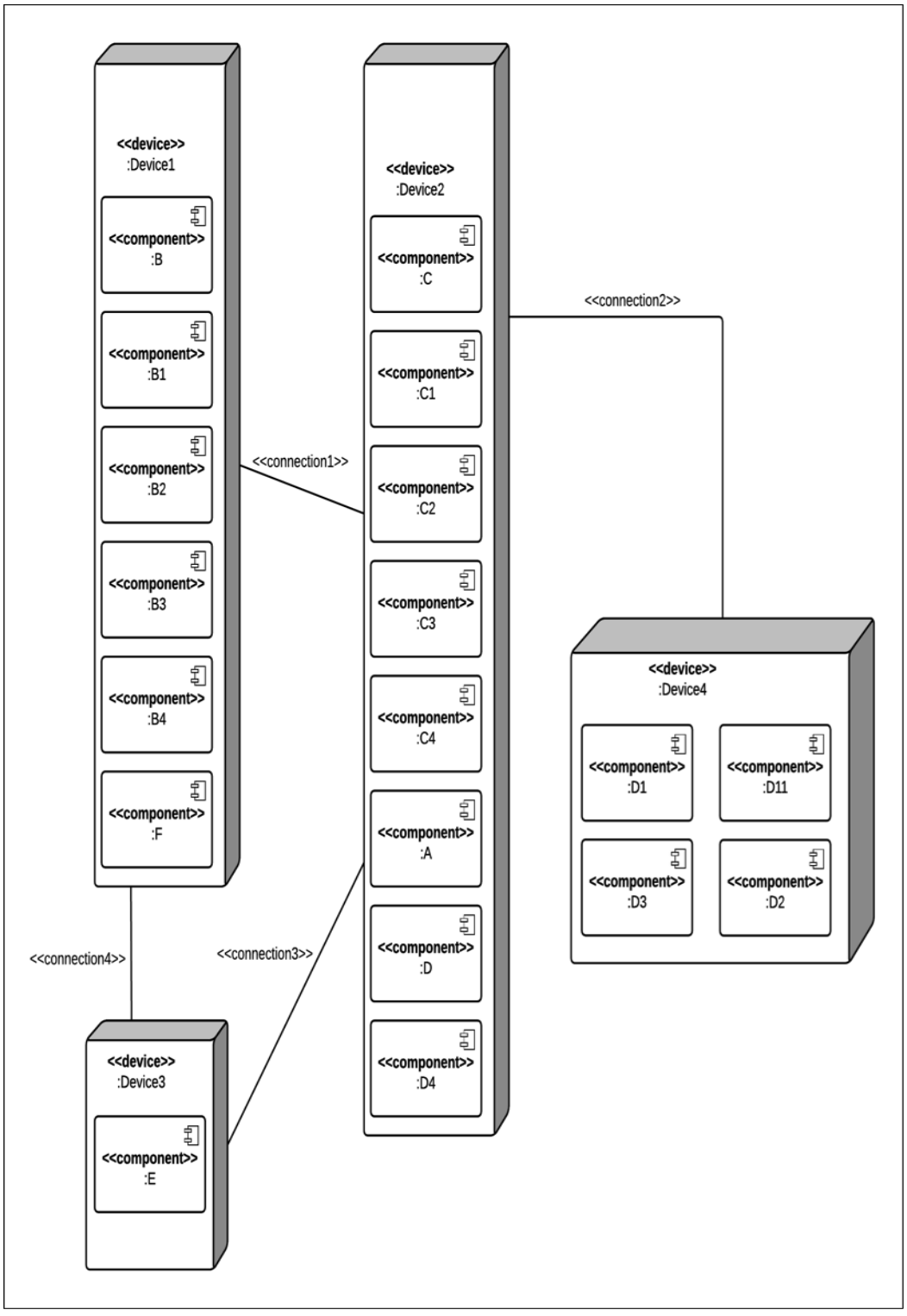

Figure 6.43: Reference architecture deployment viewpoint of the case study

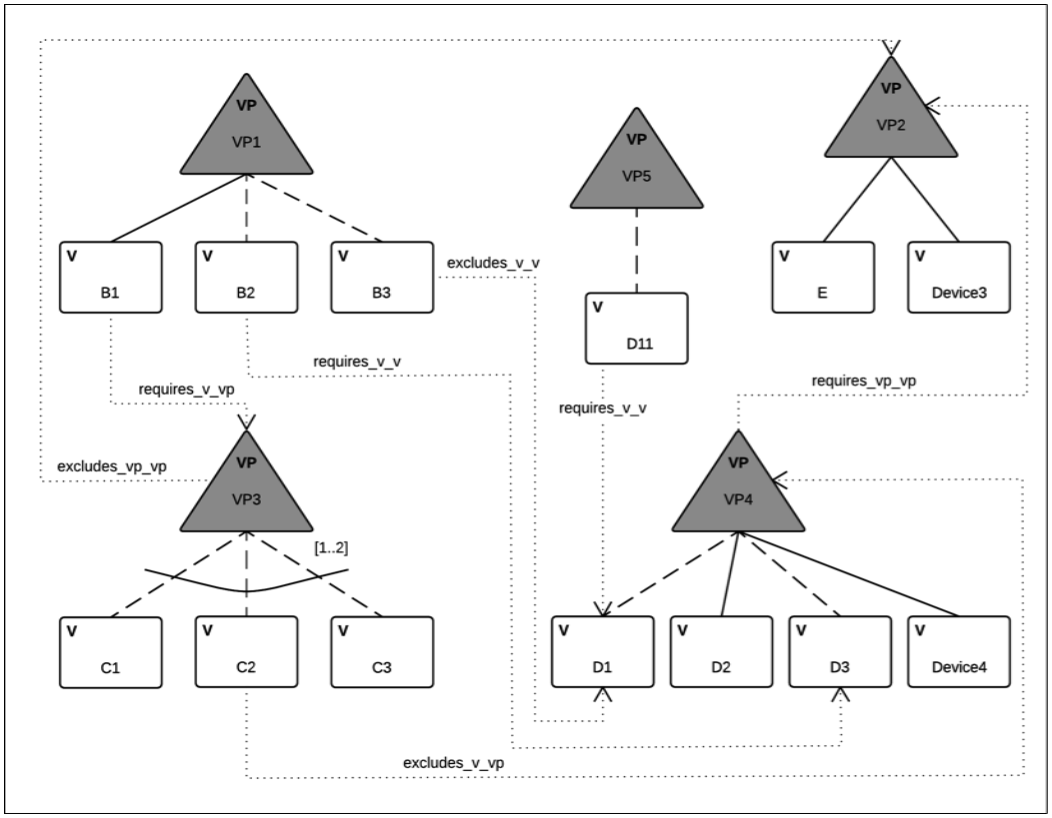

Figure 6.44: OVM of the case study for deployment viewpoint

- The component C1 on device Device2 migrates to device Device1.
- The component C2 on device Device2 migrates to device Device1.
- The component C3 on device Device2 migrates to device Device1.
- The component A on device Device2 migrates to device Device1.
- The component D2 on device Device4 migrates to device Device2.
- The component D3 on device Device4 migrates to device Device2.

It is easy to understand that, in the case study, each component is allocated to a device as default, i.e. the Allocated-to relations are defined, but some of them are set to change dynamically, which are listed above.

The two derived application architecture deployment viewpoints of the case

are demonstrated in Figure 6.45 and 6.46. The migration relations for the first application architecture are as follows:

- The copy of component B1 on device Device1 migrates to device Device2.
- The copy of component D4 on device Device2 migrates to device Device1.
- The component C2 on device Device2 migrates to device Device1.
- The component C3 on device Device2 migrates to device Device1.
- The component A on device Device2 migrates to device Device1.

The migration relations for the second application architecture are as follows:

- The execution of all components on device Device3 migrates to device Device2.
- The copy of component D1 on device Device4 migrates to device Device2.
- The copy of component D11 on device Device4 migrates to device Device2.
- The copy of component D4 on device Device2 migrates to device Device1.
- The component A on device Device2 migrates to device Device1.
- The component D2 on device Device4 migrates to device Device2.
- The component D3 on device Device4 migrates to device Device2.

In order to test our approach using the first application architecture model, we injected following faults to the model:

- A delta hardware DeviceX is added to the model and linked to the hardware Device1 with new connection5.

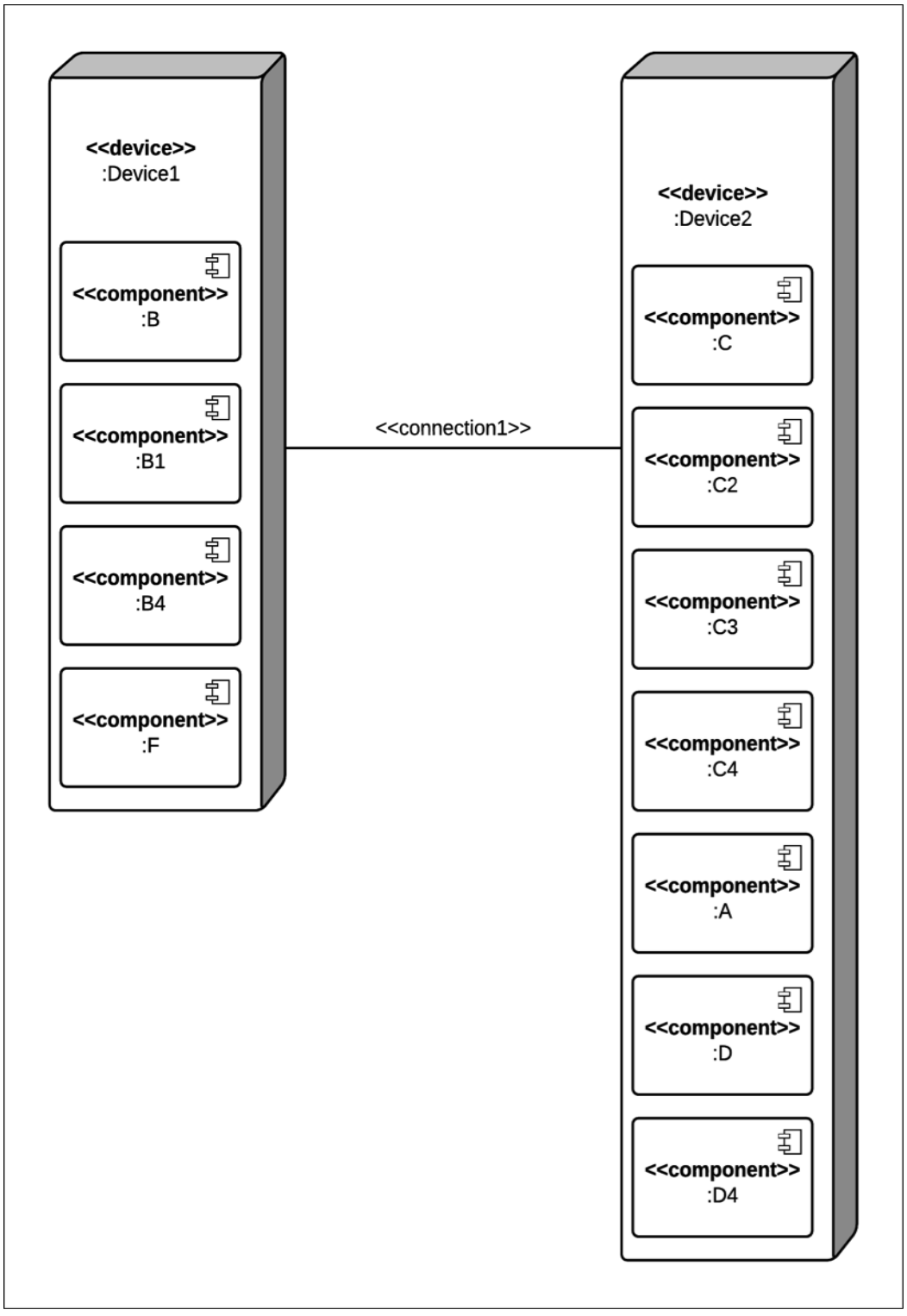

Figure 6.45: First application architecture deployment viewpoint

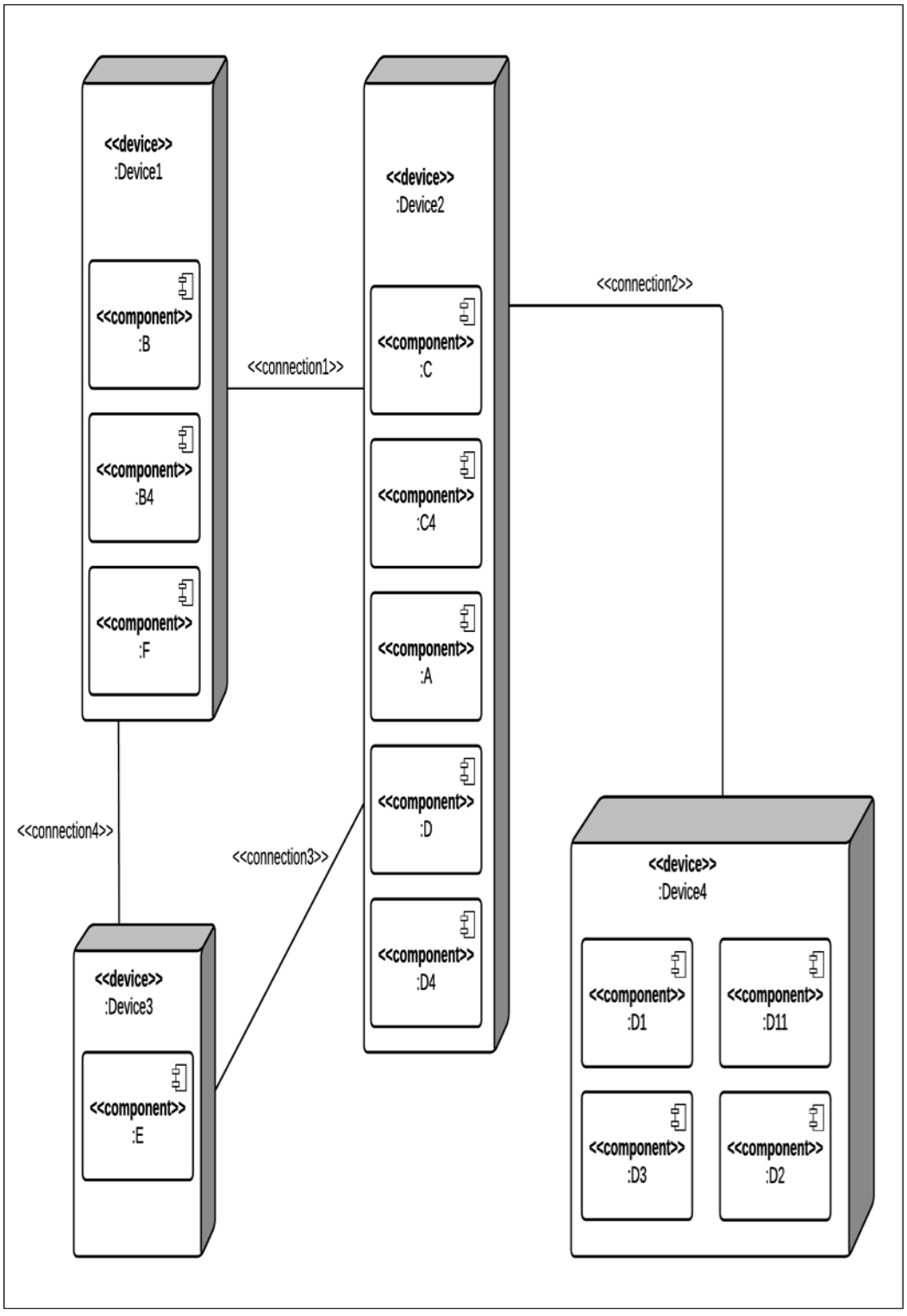

Figure 6.46: Second application architecture deployment viewpoint

- The connection link connection1 between Device1 - Device2 is altered to connection 1 between Device2 - DeviceX.
- A delta software S is added to run on Device2.
- The software C4 and C2 are removed from the model.
- The software A is moved from Device2 to Device1.

The expected output for the test using the first application architecture model is that the connection links between DeviceX and Device1, and Device2 and DeviceX are marked as Delta, the connection link between Device1 and Device2 is marked as Absence, the connection links between Device1 and Device3, Device2 and Device3, and Device2 and Device4 are marked as Not Selected Connection Link, the deployment relations between S and Device2, and A and Device1 are marked as Delta, the deployment relations between B and Device1, C and Device2, C3 and Device2, F and Device1, B4 and Device1, D and Device2, D4 and Device2, B1 and Device1, and C3 and Device2 are marked as Convergence, the deployment relations between C4 and Device2, and A and Device2 are marked as Absence, and the deployment relations between B2 and Device1, B3 and Device1, C1 and Device2, C2 and Device2, E and Device3, D1 and Device4, D3 and Device4, D2 and Device4, and D11 and Device4 are marked as Not Selected Deployment Relation. The actual output of the test can be seen in Figure 6.47. The image based reflexion model is in form of the domain mapping matrix (DMM) [56] that is used to map between two different project domains. In addition, DMM is a rectangular matrix, which is complementary to DSM. The reason why a DMM is useful for deployment viewpoint cases is that there are two different types of domain element such as software element and hardware element in a deployment viewpoint. In the DMM, columns are composed of software and hardware elements and rows are composed of only hardware elements. The color and font pattern is same as the one applied to DSMs in the other viewpoint types studied in this thesis. A “1” between a column cell and a row cell indicates that there is a deployment relation between them. Furthermore, a “2” between a column cell and a row cell indicates that there is a connection link between them, which can

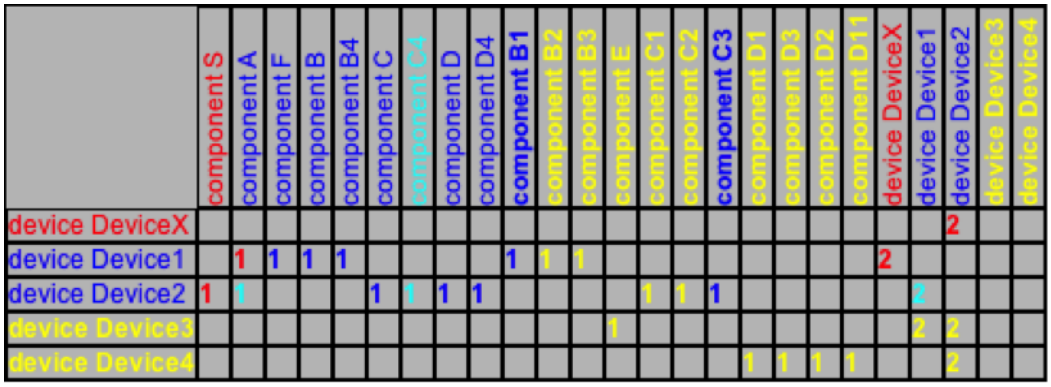

Figure 6.47: Generated image based reflexion model for the case study's deployment viewpoint model - 1

only be used for hardware elements. For instance, there is a convergent deployment relation between "component F" (source-software element) that is given on a column and "device Device1" (target-hardware element) that is given on a row. Likewise, there is a delta connection link between "device DeviceX" (source) and "device Device1" (target), for instance.

In order to test our approach using the second application architecture model, we injected following faults to the model:

- The dynamic deployment relation "The execution of all components on device Device3 migrates to device Device2" is removed from the model.
- "The component D2 on device Device4 migrates to device Device2" is altered to "The component D2 on device Device4 migrates to device Device1".
- The dynamic deployment relation "The copy of component D11 on device Device4 migrates to device Device2" is removed from the model.
- The hardware Device1 is removed from the model; hence, the connection links connection1 and connection4, and the software components F, B and B4 are removed from the model as well.
- The connection link connection3 is altered to connection6, i.e. its info is changed.

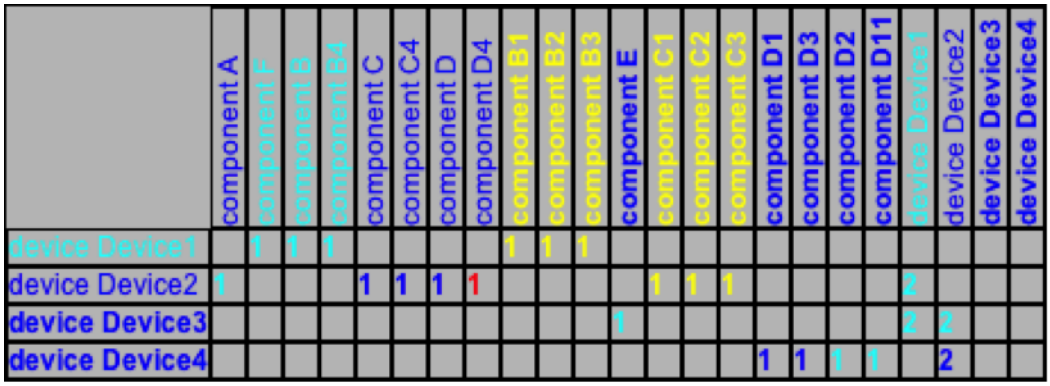

Figure 6.48: Generated image based reflexion model for the case study's deployment viewpoint model - 2

- "The copy of component D4 on device Device2 migrates to device Device1" is altered to "The copy of component D4 on device Device2 migrates to device Device4".

The expected output for the test using the second application architecture model is that the connection links between Device1 and Device2, Device1 and Device3, and Device2 and Device3 are marked as Absence, the connection link between Device2 and Device4 is marked as Convergence, the deployment relation between D4 and Device2 is marked as Delta, the deployment relations between A and Device2, B and Device1, F and Device1, B4 and Device1, E and Device3, D2 and Device4, and D11 and Device4 are marked as Absence, the deployment relations between B1 and Device1, B2 and Device1, B3 and Device1, C1 and Device2, C2 and Device2, and C3 and Device2 are marked as Not Selected Deployment Relation, and the deployment relations between C and Device2, C4 and Device2, D and Device2, D1 and Device4, and D3 and Device4 are marked as Convergence. The actual output of the test can be seen in Figure 6.48.

The way our tool decides the color of "1"s in a DMM has two major steps. It firstly checks the status of the Allocated-to deployment relations between elements since it is the main deployment relation in a deployment viewpoint and each software element must be mapped to a hardware element. Then, secondly, if there is a defined dynamic deployment relation such as Copy-migrates-to, Migrates-to or Execution-migrates-to for a software element or a hardware

element (Execution-migrates-to in this case) and its Allocated-to deployment relation is convergent, the tool evaluates the defined dynamic deployment relation's status and then, according to the result, changes the color of "1" in the DMM. If the Allocated-to deployment relation's status is something other than Convergence, the color of the "1" in the DMM is decided according to solely the Allocated-to deployment relation's status. For example, in the result of testing of our approach stated above, although there is a Convergence Allocated-to deployment relation between "component D4" and "device Device2", which means the color of "1" should be blue in Figure 6.48, it is red that means there is a Delta deployment relation between the same elements. This Delta deployment relation is the Delta Copy-migrates-to deployment relation that is result of the injected fault to the model for testing purposes (which was "The copy of component D4 on device Device2 migrates to device Device1" is altered to "The copy of component D4 on device Device2 migrates to device Device4"). Likewise, as it can be seen in Figure 6.48, the deployment relation between a software element called "component E" and hardware element called "device Device3" is marked as Absence because we injected the fault "The dynamic deployment relation 'The execution of all components on device Device3 migrates to device Device2' is removed from the model" in the testing process. This injected fault shows the idea stated above that although the Allocated-to deployment relation is preserved for a software element, if there is a defined dynamic deployment relation for that software element as well and it is not convergent, then its final color in DMM is decided according to the status of the dynamic relation. Lastly, although the information the DMM provides is limited, the generated XML file is extremely detailed and informative about the result of the conformance analysis of the deployment viewpoints.

6.2.6 Pipe-and-Filter Viewpoint

The case study's reference architecture pipe-and-filter viewpoint has 11 filters and 12 pipes in total, which is depicted in Figure 6.49. The filters are given as UML components and pipes are given as arrows between the ports of filters. For instance, there is a pipe called P1 between filter A's output port and filter F's

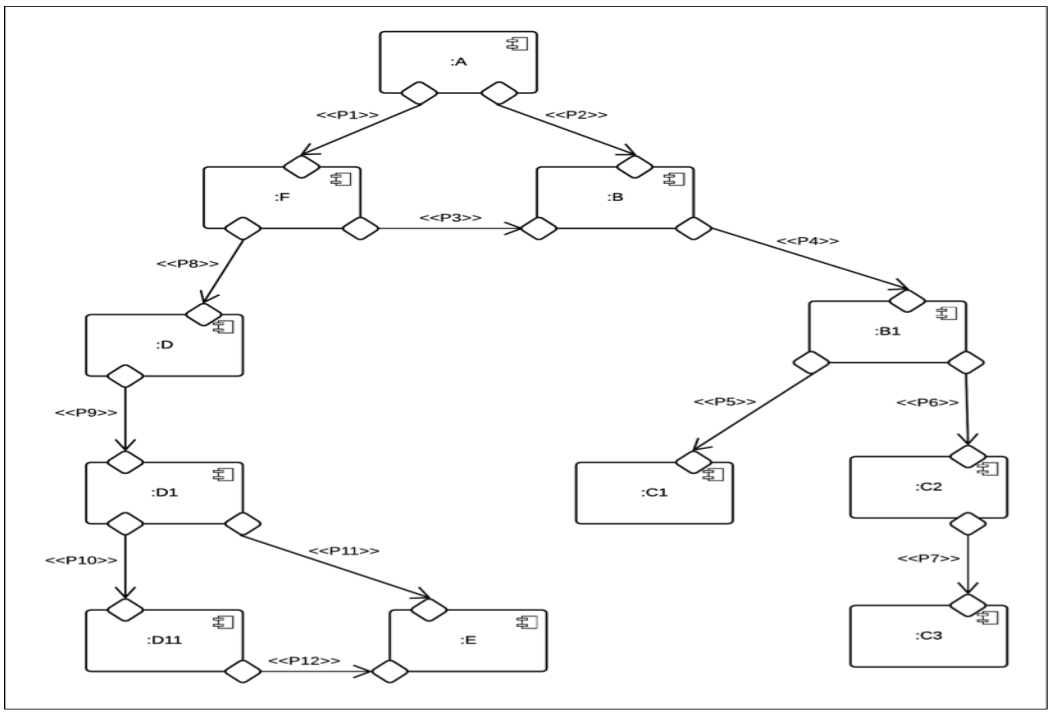

Figure 6.49: Reference architecture pipe-and-filter viewpoint of the case study

input port. For the pipe-and-filter viewpoint, we created a new OVM of the case, which can be seen in Figure 6.50. The common elements are Filter A, Filter F, Filter B and Filter D, and Pipe P1, Pipe P2, Pipe P3 and Pipe P8. The variant elements are Filter D1, Filter D11, Filter E, Filter B1, Filter C1, Filter C2 and Filter C3, and Pipe P9, Pipe P10, Pipe P11, Pipe P12, Pipe P4, Pipe P5, Pipe P6 and Pipe P7. The meaning of the variability constraints applied in the viewpoint is same as the one applied in the decomposition viewpoint.

The two derived application architecture pipe-and-filter viewpoints of the case are demonstrated in Figure 6.51 and 6.52. In order to test our approach using the first application architecture model, we injected following faults to the model:

- The common filter A is removed from the model; hence, the common pipes P1 and P2 are removed as well.
- A delta filter X is added to the model and a delta pipe PX is attached to filters X and B.

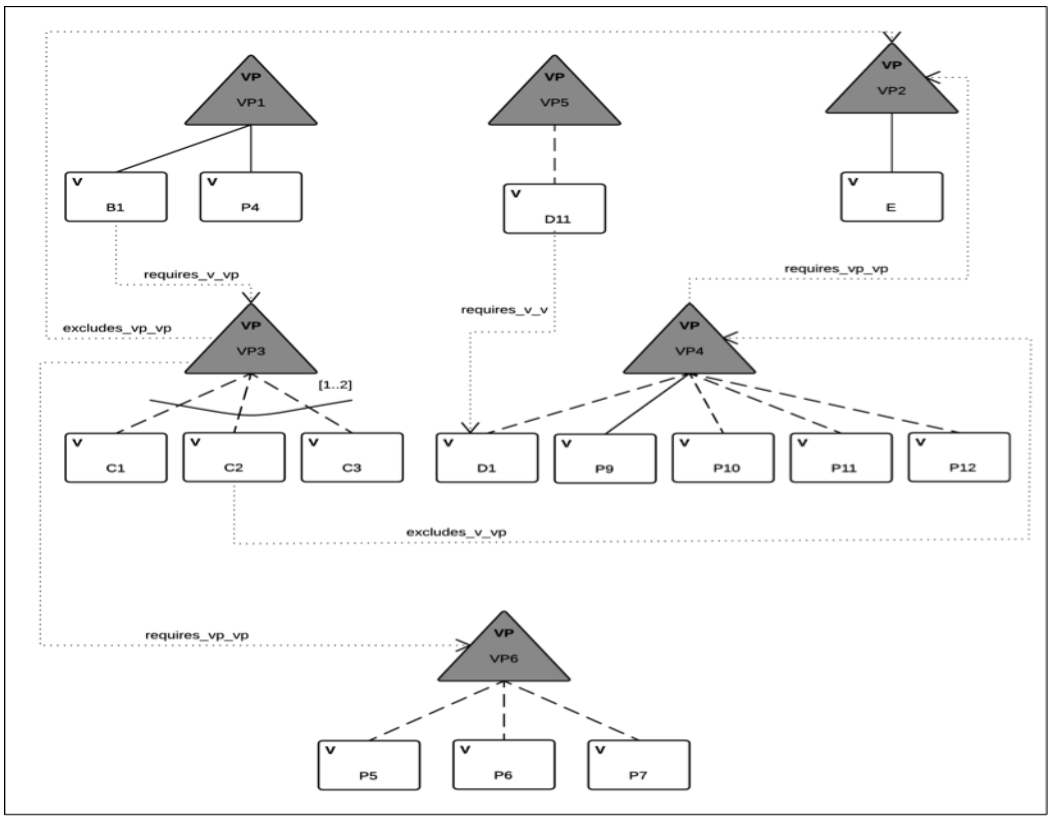

Figure 6.50: OVM of the case study for pipe-and-filter viewpoint

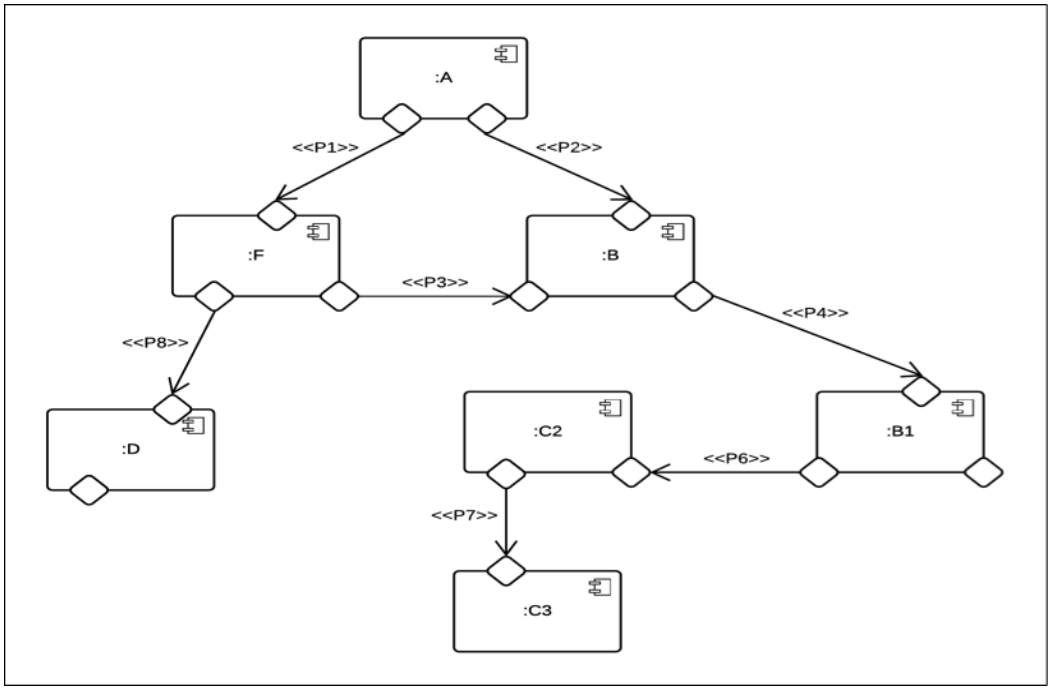

Figure 6.51: First application architecture pipe-and-filter viewpoint

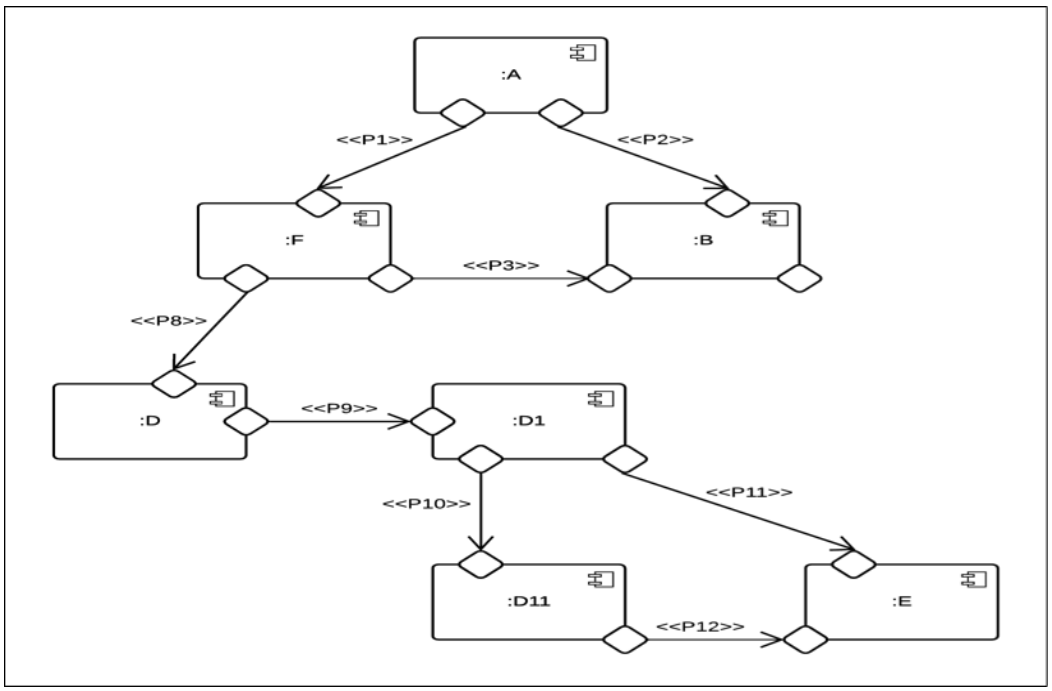

Figure 6.52: Second application architecture pipe-and-filter viewpoint

- The variant filter C3 is removed from the model; hence, the variant pipe P7 is removed as well.
- The variant pipe P6 is altered so that it is attached to filters F and B.
- The common pipe P8 is removed from the model.

The expected output for the test using the first application architecture model is that the pipe PX is marked as Delta, the pipes P1, P2, P3 and P8 are marked as Absence, the pipes P3, P4 and P6 are marked as Convergence, and the pipes P10, P11, P12, P9, P5 and P7 are marked as Not Selected Variant, the filter X is marked as Delta, the filter A is marked as Absence, the filters F, B, D, B1 and C2 are marked as Convergence, and the filters E, C1, C3, D1 and D11 are marked as Not Selected Variant, the attachment relations PX-X (which means the pipe PX is attached to filter X) and PX-B are marked as Delta, the attachment relations P1-A, P1-F, P2-A, P2-B, P8-F, P8-D, P6-B1 and P6-C2 are marked as Absence, the attachment relations P3-F, P3-B, P4-B and P4-B1 are marked as Convergence, and the attachment relations P10-D1, P10-D11, P11-E, P11-D1,

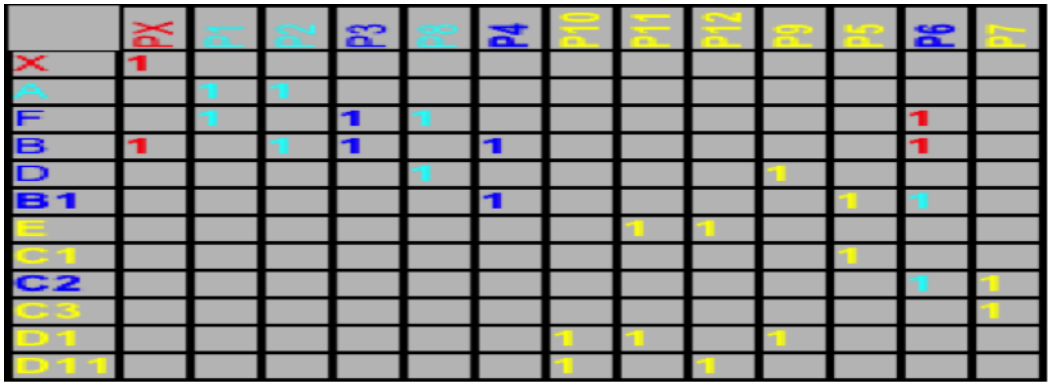

Figure 6.53: Generated image based reflexion model for the case study's pipe-and-filter viewpoint model - 1

P12-E, P12-D11, P9-D, P9-D1, P5-B1, P5-C1, P7-C2 and P7-C3 are marked as Not Selected Attachment Relation. The actual output of the test can be seen in Figure 6.53. The image based reflexion model is in the form of the DMM in which the columns are composed of pipes and the rows are composed of filters.

In addition, the color and font pattern is same as the one applied to DSMs in the other viewpoint types studied in this thesis. A "1" between a column cell and a row cell indicates that there is an attachment relation between them. For example, in Figure 6.53, the blue "1"s between convergent pipe P3 and convergent filters F and B highlight that there is a convergent attachment relation between the pipe and the two filters that are placed on the right and left sides of the pipe.

In order to test our approach using the second application architecture model, we injected following faults to the model:

- The variant pipe P9 is removed from the model.
- The common pipe P3 is removed from the model.
- The variant pipe P12 is altered so that it is attached to filters D1 and D11.
- The variant filter E is removed from the model, which means a variability constraint is violated (see the OVM); hence, the variant pipe P11 is removed as well.

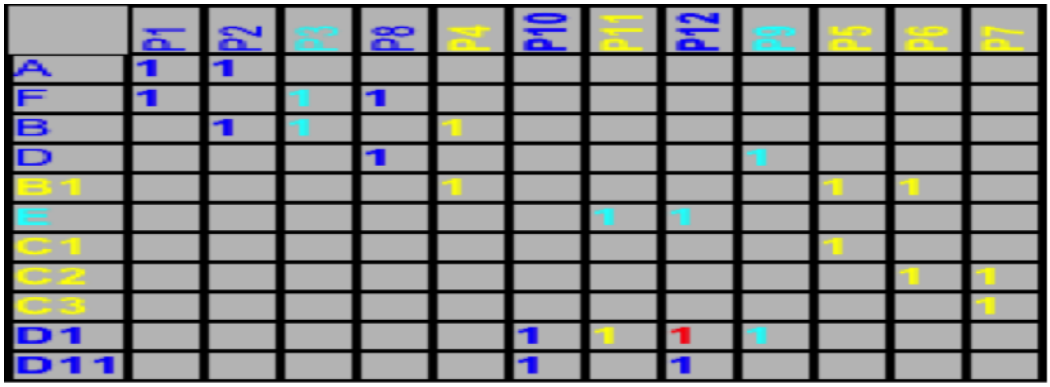

Figure 6.54: Generated image based reflexion model for the case study's pipe-and-filter viewpoint model - 2

The expected output for the test using the second application architecture model is that the pipes P1, P2, P8, P10 and P12 are marked as Convergence, the pipes P3 and P9 are marked as Absence, the pipes P4, P11, P5, P6 and P7 are marked as Not Selected Variant, the filters A, F, B, D, D1 and D11 are marked as Convergence, the filter E is marked as Absence, and the filters B1, C1, C2 and C3 are marked as Not Selected Variant, the attachment relations P1-A, P1-F, P2-A, P2-B, P8-F, P8-D, P10-D1, P10-D11 and P12-D11 are marked as Convergence, the attachment relation P12-D1 is marked as Delta, the attachment relations P3-F, P3-B, P11-E, P12-E, P9-D and P9-D1 are marked as Absence, and the attachment relations P4-B, P4-B1, P11-D1, P5-B1, P5-C1, P6-B1, P6-C2, P7-C2 and P7-C3 are marked as Not Selected Attachment Relation. The actual output of the test can be seen in Figure 6.54.

6.3 Discussion

In order to test our approach, with respect to the ACCC defined for each viewpoint type, we totally injected 53 faults to the two different derived application architecture models. We aimed to test all possible cases without repeating same scenario on a viewpoint type. However, it is noteworthy to state that since each viewpoint model shares variability part in common, we did not include the faults that test it into the set of faults that is specific to each viewpoint type; we used

Table 6.1: Test results for each viewpoint type

Viewpoint Type	**Number of Faults Injected**		**Number of Faults Detected**	
	1st App.	**2nd App.**	**1st App.**	**2nd App.**
Decomposition	3	5	3	5
Uses	5	4	5	4
Generalization	7	2	7	2
Layered	5	2	5	2
Deployment	5	6	5	6
Pipe-and-Filter	5	4	5	4

them in testing of the decomposition viewpoint instead. Table 6.1 lists related results considering the viewpoint type and application architecture models. For example, our tool detected the all injected 7 faults to the first application generalization viewpoint model and the effects they cause represented on the generated reflexion model.

Lastly, we should underline that input models that will be used in our tool have to conform to their defined metamodels by obeying constraints the metamodels set. Otherwise, the tool does not run correctly and, most likely, it will give an error.

Chapter 7

Related Work

Software architecture conformance analysis techniques have been proposed in the software engineering literature. Kazman et al. [57] propose a scenario-based approach for analyzing software architectures, which is called "Software Architecture Analysis Method" (SAAM). SAAM uses a set of scenarios to evaluate architectures by taking defined quality attributes into account. In another work [58], a technique called "Architecture Tradeoff Analysis Method" (ATAM) is proposed to test conformance of software architectures to multiple quality attributes such as performance, modifiability and security. ATAM is an iterative method that also uses architectural views before the step of realization of scenarios in the general process of the approach. The work [59] is an experience report in the context of SPLE, which uses a combined method including SAAM and ATAM for analyzing a particular product line. Lutz and Gannod [59] state that, in their approach, the required scenario by SAAM is the product line's usage description and that the necessary quality attributes for ATAM are specific to product line architectures, e.g. extensibility of variations, portability of sub-products, etc. In [60], "Holistic Product Line Architecture Assessment" (HoPLAA) technique is proposed, which extends ATAM for evaluating the product line architecture (PLA) as the first step and single product architecture (PA) as the second step. Since HoPLAA is an extension of ATAM, the method directly focuses on quality attribute tradeoffs for architectures in the product line. Oliveira Junior et al.

[61] introduce a systematic approach to assess UML-based software product line architectures considering its quality attributes. It can be stated that, by taking these approaches into consideration, the architecture conformance analysis process in the context of SPLE is mostly focused on the conformance of architecture to its quality attributes rather than the conformance of architecture to another architecture.

In terms of architecture conformance testing in SPLE, Muccini and van der Hoek [62] point out that, for checking the conformance of architecture to corresponding code, there are two cases that the tester must aware of when defining the conformance condition. It is counted as conformance of the code to the PLA if the code conforms to a PA or conforms to all the available PAs derived from the PLA [62]. In addition, in [62], it is stated that it is possible to check conformance of the derived PAs to the PLA by considering whether all structural and behavioral constraints defined in the PLA are met by PAs or not.

The study by Tekinerdogan et al. [14] extends the general reflexion modeling idea to the context of multiple product line engineering for architecture conformance analysis process. The work categorizes the conformance analysis approaches in two: single system conformance checking and system-of-systems conformance checking. In the former, a single product line is used to develop software products. In the latter, on the other hand, multiple different product lines are active, which generate intermediate products that are used in the development of final software products. In addition, the work considers architecture conformance analysis on both architectural level and architecture-code level for the two categories.

In terms of tool support for architecture conformance analysis, there are a couple of tools developed. Sotograph [63] is a tool for architecture analysis and finding differences between a software system's different versions using a software repository. Lattix Architect [64] is based on DSMs to check software architecture conformance. However, these tools are not designed for SPLE. A tool called SAVE [65] is capable of conformance checking with structural views and behavioral views. In terms of SPLE it supports a method of variant comparison, which

informs the user about the implemented variability in the system, rather than a full architecture conformance analysis technique.

Chapter 8

Conclusion

Software product lines have become a core development approach for large-scale organizations that aim at improved productivity and increased quality in the development of a family of software systems. However, the main benefits of product lines can evaporate if the conformance between assets developed in the domain engineering phase and the ones developed in the application engineering phase is not maintained. This case is very likely in the architectural level where application architectures' conformance to reference architecture is crucial and can easily result in a drift between each other if it is not checked regularly and carefully.

In order to detect possible drifts occurred between reference architecture and derived application architectures in product lines, this work presented a unique model-driven approach that handles architecture conformance analysis using architecture viewpoints and produces corresponding results as a reflexion model in terms of both visually and textually. Furthermore, by using the architecture viewpoints in the approach, we had an opportunity of considering different stakeholders' concerns; hence, the approach is applicable to use in conformance analysis for various aspects of software architectures defined in the product line.

It is our hope that this study contributes to the work of other researchers

in this domain and also, with the results presented in the conducted systematic literature review, encourages those who have a passion to initiate new research activities related to this domain.

As a limitation of this work, it can be counted that the approach, indirectly the developed tool as well, should be tested on a large-scale industrial case. However, it must be noted that the approach did great on the small case study.

One of our future works is that we will modify our approach and tool to be applicable for other viewpoint frameworks. This idea seems feasible since all viewpoint frameworks have some points in common and existing variability can be handled. The other future work we plan to carry out is to enhance our approach so that it can be used for analysis of architecture to code conformance in product lines.

Bibliography

[1] *Recommended Practice for Architectural Description of Software-Intensive Systems (ISO/IEC 42010)*, July 2007.

[2] J.-C. Royer and H. Arboleda, *Model-Driven and Software Product Line Engineering (ISTE)*. Wiley-IEEE Press, 1st ed., 2012.

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[4] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd ed., 2010.

[5] B. Tekinerdogan and H. Sözer, "Defining architectural viewpoints for quality concerns," in *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*, pp. 26–34, 2011.

[6] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 364–380, Apr. 2001.

[7] "Eclipse epsilon." http://www.eclipse.org/epsilon/ [26 November 2015].

[8] "Apache ant." https://ant.apache.org/ [28 November 2015].

[9] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[10] S. J. Mellor, S. Kendall, A. Uhl, and D. Weise, *MDA Distilled*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[11] "Omg, object management group." http://www.omg.org [26 November 2015].

[12] "Model-to-model transformation." https://www.eclipse.org/mmt/ [26 November 2015].

[13] B. Tekinerdogan, "Software architecture." chapter in: T. Gonzalez, J.L. Diaz-Herrera, Computer Science Handbook, Second Edition, Volume I: Computer Science and Software Engineering, 2014.

[14] B. Tekinerdogan, E. Çilden, Ö. Ö. Erdogan, and O. Aktug, "Architecture conformance analysis approach within the context of multiple product line engineering," in *23rd Australian Software Engineering Conference, ASWEC 2014, Milsons Point, Sydney, NSW, Australia, April 7-10, 2014*, pp. 25–28, 2014.

[15] Y. Ma, J. Chen, and J. Wu, "Research on the phenomenon of software drift in software processes," in *IWPSE*, pp. 195–198, IEEE Computer Society, 2005.

[16] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, (Washington, DC, USA), pp. 12–, IEEE Computer Society, 2007.

[17] L. Passos, R. Terra, M. Valente, R. Diniz, and N. Mendonça, "Static architecture-conformance checking: An illustrative overview," *Software, IEEE*, vol. 27, pp. 82–89, Sept 2010.

[18] R. Kauppinen, "Testing framework-based software product lines," Master's thesis, R. Kauppinen, 2003.

[19] M. Coteli, "Testing effectiveness and effort in software product lines," Master's thesis, Middle East Technical University, 2013.

[20] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, (Washington, DC, USA), pp. 459–468, IEEE Computer Society, 2010.

[21] A. Sawant, P. Bari, and P. Chawan, "Software testing techniques and strategies," *International Journal of Engineering Research and Applications (IJERA)*, vol. 2, no. 3, pp. 980–986, 2012.

[22] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," no. EBSE 2007-001, 2007.

[23] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, (Swinton, UK, UK), pp. 68–77, British Computer Society, 2008.

[24] F. van der Linden, "Software product families in europe: The esaps & cafe projects," *IEEE Softw.*, vol. 19, pp. 41–49, July 2002.

[25] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, (Washington, DC, USA), pp. 45–54, IEEE Computer Society, 2001.

[26] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Sci. Comput. Program.*, vol. 53, pp. 333–352, Dec. 2004.

[27] M. F. Johansen, Ø. Haugen, and F. Fleurey, "Properties of realistic feature models make combinatorial testing of product lines feasible," in *MODELS*, pp. 638–652, Springer, 2011.

[28] P. Heymans and J. Trigaux, "Software product lines: State of the art," tech. rep., Project of Product Line ENgineering of food TraceabilitY software, First Europe Objectif 3, 2003.

[29] A. B. Marques, R. Rodrigues, and T. Conte, "Systematic literature reviews in distributed software development: A tertiary study.," in *ICGSE*, pp. 134–143, IEEE Computer Society, 2012.

[30] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," *Inf. Softw. Technol.*, vol. 51, pp. 7–15, Jan. 2009.

[31] S. Imtiaz, M. Bano, N. Ikram, and M. Niazi, "A tertiary study: Experiences of conducting systematic literature reviews in software engineering," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, (New York, NY, USA), pp. 177–182, ACM, 2013.

[32] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, (Washington, DC, USA), pp. 273–281, IEEE Computer Society, 2004.

[33] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Inf. Softw. Technol.*, vol. 53, pp. 625–637, June 2011.

[34] "dblp, computer science bibliography." http://dblp.uni-trier.de/db/ [11 March 2015].

[35] B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering - a tertiary study," *Information and Software Technology*, vol. 52, no. 8, pp. 792 – 805, 2010.

[36] J. M. Verner, O. P. Brereton, B. A. . Kitchenham, M. Turner, and M. Niazi, "Systematic literature reviews in global software development: A tertiary study," in *Evaluation and Assessment in Software Engineering (EASE 2012)*, pp. 2 – 11, IET, 2012.

[37] B. Kitchenham, "Procedures for performing systematic reviews," keele university. technical report tr/se-0401, Department of Computer Science, Keele University, UK, 2004.

[38] D. Budgen, M. Turner, P. Brereton, and B. Kitchenham, "Using Mapping Studies in Software Engineering," in *Proceedings of PPIG 2008*, pp. 195–204, Lancaster University, 2008.

[39] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Computer Society 2004 Guide, 2004.

[40] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 85–103, IEEE Computer Society, 2007.

[41] "Omg, meta object facility (mof) 2.0 query/view/transformation (qvt)." http://www.omg.org/spec/QVT/ [25 June 2015].

[42] "Omg, mof model to text transformation language 1.0." http://www.omg.org/spec/MOFM2T/1.0/ [25 June 2015].

[43] I. Reinhartz-Berger and K. Figl, "Comprehensibility of orthogonal variability modeling languages: The cases of cvl and ovm," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, (New York, NY, USA), pp. 42–51, ACM, 2014.

[44] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey," in *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010*, pp. 374–379, 2010.

[45] "Emfatic, a textual syntax for emf ecore metamodels." https://www.eclipse.org/emfatic/ [4 October 2015].

[46] "Emf, eclipse modeling framework." https://eclipse.org/modeling/emf/ [4 October 2015].

[47] "Hutn, human usable textual notation." http://www.eclipse.org/epsilon/doc/hutn/ [4 October 2015].

[48] "Evl, epsilon validation language." http://www.eclipse.org/epsilon/doc/evl/ [4 October 2015].

[49] "Xml, extensible markup language." http://www.w3.org/XML/ [4 October 2015].

[50] "Egl, epsilon generation language." http://www.eclipse.org/epsilon/doc/egl/ [4 October 2015].

[51] F. Roos-Frantz, D. Benavides, and A. Ruiz-Cortés, "Feature model to orthogonal variability model transformation towards interoperability between tools," *Knowledge Industry Survival Strategy Initiative*, 2009.

[52] "Ocl, object constraint language." http://www.omg.org/spec/OCL/ [5 October 2015].

[53] E. Demirli, "Model-driven engineering of software architecture viewpoints," Master's thesis, Department of Computer Engineering and the Graduate School of Engineering and Science of Bilkent University, 2012.

[54] B. Tekinerdogan and E. Demirli, "Evaluation framework for software architecture viewpoint languages," in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, (New York, NY, USA), pp. 89–98, ACM, 2013.

[55] E. Demirli and B. Tekinerdogan, "Software language engineering of architectural viewpoints," in *Proceedings of the 5th European Conference on Software Architecture (ECSA 2011)*, pp. 336 – 343, Springer Berlin Heidelberg, 2011.

[56] B. Tekinerdogan, "Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices," 2015.

[57] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *IEEE Softw.*, vol. 13, pp. 47–55, Nov. 1996.

[58] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and S. Carriere, "The architecture tradeoff analysis method," Tech. Rep. CMU/SEI-98-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1998.

[59] R. R. Lutz and G. C. Gannod, "Analysis of a software product line architecture: an experience report.," *Journal of Systems and Software*, vol. 66, no. 3, pp. 253–267, 2003.

[60] F. Olumofin and V. Misic, "Extending the atam architecture evaluation to product line architectures," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pp. 45–56, 2005.

[61] E. A. O. Junior, I. M. S. Gimenes, J. C. Maldonado, P. C. Masiero, and L. Barroca, "Systematic evaluation of software product line architectures," *Journal of Universal Computer Science*, vol. 19, no. 1, pp. 25–52, 2013.

[62] H. Muccini and A. van der Hoek, "Towards testing product line architectures," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 99 – 109, 2003. TACoS'03, International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of {ETAPS} 2003).

[63] "Sotograph." http://www.hello2morrow.com/products/sotograph/ [5 December 2015].

[64] "Lattix architect." http://lattix.com/lattix-architect-and-lattix-analyst/ [5 December 2015].

[65] S. Duszynski, J. Knodel, and M. Lindvall, "Save: Software architecture visualization and evaluation," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pp. 323–324, March 2009.

Appendix A

Databases and Venues used in Search Process

Databases subject to automated search:

- *ACM Digital Library*
- *CiteSeerX*
- *Google Scholar*
- *IEEE Explore*
- *ScienceDirect*
- *Scopus*
- *Springer Link*
- *ISI Web of Knowledge*
- *Wiley InterScience Online Library*

Venues subject to manual search:

- Journals
  - *IEEE TSE - Transactions on Software Engineering*
  - *IST - Information and Software Technology*
  - *STVR - Software Testing, Verification and Reliability*
- Conferences
  - *ICSE - Software Engineering*
  - *EASE - Evaluation and Assessment in Software Engineering*
  - *ICSOFT-EA - Software Engineering and Applications*
  - *ICSOFT-PT - Software Paradigm Trends*
  - *ICST - Software Testing*
  - *ICTSS - Testing Software and Systems*
  - *ISSTA - Software Testing and Analysis*
  - *SPLC - Software Product Line Conference*
- Workshops
  - *PLEASE - Product Line Approaches in Software Engineering*
  - *SPLiT - Software Product Lines Testing*

Appendix B

Search Strings

- *ACM Digital Library*

  (Title:"testing" OR Title:"test") AND (Title:"software product line" OR Title:"software product families" OR Title:"software product lines" OR Title:"product line" OR Title:"product lines" OR Title:"software product family" OR Title:"system family" OR Title:"system families" OR Title:"software product line engineering" OR Title:"product line engineering") AND (Title:"review of studies" OR Title:"structured review" OR Title:"systematic review" OR Title:"literature review" OR Title:"literature analysis" OR Title:"in-depth survey" OR Title:"literature survey" OR Title:"meta analysis" OR Title:"analysis of research" OR Title:"empirical body of knowledge" OR Title:"overview of existing research" OR Title:"systematic literature review" OR Title:"SLR" OR Title:"mapping studies" OR Title:"tertiary study" OR Title:"mapping study")

  (Abstract:"testing" OR Abstract:"test") AND (Abstract:"software product line" OR Abstract:"software product families" OR Abstract:"software product lines" OR Abstract:"product line" OR Abstract:"product lines" OR Abstract:"software product family" OR Abstract:"system family" OR Abstract:"system families" OR Abstract:"software product line engineering" OR Abstract:"product line engineering") AND (Abstract:"review of

studies" OR Abstract:"structured review" OR Abstract:"systematic review" OR Abstract:"literature review" OR Abstract:"literature analysis" OR Abstract:"in-depth survey" OR Abstract:"literature survey" OR Abstract:"meta analysis" OR Abstract:"analysis of research" OR Abstract:"empirical body of knowledge" OR Abstract:"overview of existing research" OR Abstract:"systematic literature review" OR Abstract:"SLR" OR Abstract:"mapping studies" OR Abstract:"tertiary study" OR Abstract:"mapping study")

(Keywords:"testing" OR Keywords:"test") AND (Keywords:"software product line" OR Keywords:"software product families" OR Keywords:"software product lines" OR Keywords:"product line" OR Keywords:"product lines" OR Keywords:"software product family" OR Keywords:"system family" OR Keywords:"system families" OR Keywords:"software product line engineering" OR Keywords:"product line engineering") AND (Keywords:"review of studies" OR Keywords:"structured review" OR Keywords:"systematic review" OR Keywords:"literature review" OR Keywords:"literature analysis" OR Keywords:"in-depth survey" OR Keywords:"literature survey" OR Keywords:"meta analysis" OR Keywords:"analysis of research" OR Keywords:"empirical body of knowledge" OR Keywords:"overview of existing research" OR Keywords:"systematic literature review" OR Keywords:"SLR" OR Keywords:"mapping studies" OR Keywords:"tertiary study" OR Keywords:"mapping study")

- *CiteSeerX*

  title:("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping

studies" OR "tertiary study" OR "mapping study")

abstract:("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study")

keyword:("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study")

- *Google Scholar*

  ("testing" OR "test") AND ("product line" OR "software product line" OR "software family") AND ("systematic literature review" OR "mapping study" OR "meta analysis" OR "tertiary study")

- *IEEE Explore*

  ("Document Title":"testing" OR "Document Title":"test") AND ("Document Title":"software product line" OR "Document Title":"software product families" OR "Document Title":"software product lines" OR "Document Title":"product line" OR "Document Title":"product lines"

OR "Document Title":"software product family" OR "Document Title":"system family" OR "Document Title":"system families" OR "Document Title":"software product line engineering" OR "Document Title":"product line engineering") AND ("Document Title":"review of studies" OR "Document Title":"structured review" OR "Document Title":"systematic review" OR "Document Title":"literature review" OR "Document Title":"literature analysis" OR "Document Title":"in-depth survey" OR "Document Title":"literature survey" OR "Document Title":"meta analysis" OR "Document Title":"analysis of research" OR "Document Title":"empirical body of knowledge" OR "Document Title":"overview of existing research" OR "Document Title":"systematic literature review" OR "Document Title":"SLR" OR "Document Title":"mapping studies" OR "Document Title":"tertiary study" OR "Document Title":"mapping study")

("Abstract":"testing" OR "Abstract":"test") AND ("Abstract":"software product line" OR "Abstract":"software product families" OR "Abstract":"software product lines" OR "Abstract":"product line" OR "Abstract":"product lines" OR "Abstract":"software product family" OR "Abstract":"system family" OR "Abstract":"system families" OR "Abstract":"software product line engineering" OR "Abstract":"product line engineering") AND ("Abstract":"review of studies" OR "Abstract":"structured review" OR "Abstract":"systematic review" OR "Abstract":"literature review" OR "Abstract":"literature analysis" OR "Abstract":"in-depth survey" OR "Abstract":"literature survey" OR "Abstract":"meta analysis" OR "Abstract":"analysis of research" OR "Abstract":"empirical body of knowledge" OR "Abstract":"overview of existing research" OR "Abstract":"systematic literature review" OR "Abstract":"SLR" OR "Abstract":"mapping studies" OR "Abstract":"tertiary study" OR "Abstract":"mapping study")

("Author Keywords":"testing" OR "Author Keywords":"test") AND ("Author Keywords":"software product line" OR "Author Keywords":"software product families" OR "Author Keywords":"software product lines" OR "Author Keywords":"product line" OR "Author Keywords":"product lines"

OR "Author Keywords":"software product family" OR "Author Keywords":"system family" OR "Author Keywords":"system families" OR "Author Keywords":"software product line engineering" OR "Author Keywords":"product line engineering") AND ("Author Keywords":"review of studies" OR "Author Keywords":"structured review" OR "Author Keywords":"systematic review" OR "Author Keywords":"literature review" OR "Author Keywords":"literature analysis" OR "Author Keywords":"in-depth survey" OR "Author Keywords":"literature survey" OR "Author Keywords":"meta analysis" OR "Author Keywords":"analysis of research" OR "Author Keywords":"empirical body of knowledge" OR "Author Keywords":"overview of existing research" OR "Author Keywords":"systematic literature review" OR "Author Keywords":"SLR" OR "Author Keywords":"mapping studies" OR "Author Keywords":"tertiary study" OR "Author Keywords":"mapping study")

- *ScienceDirect*

  tak(("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study"))

- *Scopus*

  TITLE-ABS-KEY(("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line

engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study"))

- *Springer Link*

  ("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in-depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study")

- *ISI Web of Knowledge*

  (TS="testing" OR TS="test") AND (TS="software product line" OR TS="software product families" OR TS="software product lines" OR TS="product line" OR TS="product lines" OR TS="software product family" OR TS="system family" OR TS="system families" OR TS="software product line engineering" OR TS="product line engineering") AND (TS="review of studies" OR TS="structured review" OR TS="systematic review" OR TS="literature review" OR TS="literature analysis" OR TS="in-depth survey" OR TS="literature survey" OR TS="meta analysis" OR TS="analysis of research" OR TS="empirical body of knowledge" OR TS="overview of existing research" OR TS="systematic literature review" OR TS="SLR" OR TS="mapping studies" OR TS="tertiary study" OR TS="mapping study")

- *Wiley InterScience Online Library*

("testing" OR "test") AND ("software product line" OR "software product families" OR "software product lines" OR "product line" OR "product lines" OR "software product family" OR "system family" OR "system families" OR "software product line engineering" OR "product line engineering") AND ("review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "literature analysis" OR "in depth survey" OR "literature survey" OR "meta analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "systematic literature review" OR "SLR" OR "mapping studies" OR "tertiary study" OR "mapping study") AND "software"

Appendix C

List of Selected Secondary Studies

A) B.P. Lamancha, M. Polo, M. Piattini, Systematic Review on Software Product Line Testing, in: Software and Data Technologies, Communications in Computer and Information Science, Vol. 170, Springer Berlin Heidelberg, 2013, pp. 58-71, doi:10.1007/978-3-642-29578-2_4.

B) E. Engstrom, P. Runeson, Software Product Line Testing - A Systematic Mapping Study, Information and Software Technology, Vol. 53, Issue 1, 2011, pp. 2-13, doi:10.1016/j.infsof.2010.05.011.

C) I.d.C. Machado, J.D. McGregor, Y.C. Cavalcanti, E.S.d. Almeida, On Strategies for Testing Software Product Lines: A Systematic Literature Review, Information and Software Technology, Vol. 56, Issue 10, 2014, pp. 1183-1199, doi:10.1016/j.infsof.2014.04.002.

D) P.A.d.M.S. Neto, I.d.C. Machado, J.D. McGregor, E.S.d. Almeida, S.R.d.L. Meira, A Systematic Mapping Study of Software Product Line Testing, Information and Software Technology, Vol. 53, Issue 5, 2011, pp. 407-423, doi:10.1016/j.infsof.2010.12.003.

Appendix D

Quality Assessment Result

Study Reference	QA1 (selection criteria)	QA2 (literature search)	QA3 (quality assessment)	QA4 (studies description)	Total Score
A	Y	P	N	Y	2.5
B	Y	N	N	Y	2
C	Y	Y	Y	P	3.5
D	Y	Y	Y	P	3.5

Appendix E

Data Extraction Form

	Extraction Element	Content
1	Paper Reference	A pointer to refer the paper such as A, P1, etc.
2	Title	The paper's full title
3	Year	The year when the paper was published
4	Author(s)	
5	Repository	The digital library the paper was downloaded from
6	Source	The conference or journal
7	Publication Venue(s)	Publication venues' name
8	Type	SLR or MS
9	Scope	Research trends or specific research question
10	Main SPL Testing Topic	Testing levels, test process, etc.
11	Research Question(s)	
12	Research Objective(s)	
13	Study Guideline(s)	Whether the paper referenced an EBSE paper or the SLR guidelines, and their names
14	Number of Primary Studies Included	The number of primary studies used in the SLR/MS
15	Summary	Summary of the paper
16	Quality Assessment	The quality score of the study
17	Literature Search Type(s)	Digital, manual, etc.
18	Reviewer Note	The reviewer's own opinions about the study
19	Additional Note	Publication details

Appendix F

References of Studies Cited in the Reviewed Secondary Studies

[A1] J.D. McGregor, Testing a Software Product Line, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2001-TR-022, 2001.

[A2] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, B. Medina, Architecture-Based Unit Testing of the Flight Software Product Line, in: Proceeding SPLC 2010 Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, Vol. 6287, Springer Berlin Heidelberg, 2010, pp. 256-270, doi:10.1007/978-3-642-15579-6_18.

[A3] E. Uzuncaova, S. Khurshid, D. Batory, Incremental Test Generation for Software Product Lines, IEEE Transactions on Software Engineering, Vol. 36, Issue 3, IEEE, 2010, pp. 309-322, doi:10.1109/TSE.2010.30.

[A4] A. Tevanlinna, J. Taina, R. Kauppinen, Product Family Testing: A Survey, ACM SIGSOFT Software Engineering Notes, Vol.29, Issue 2, ACM, 2004, pp. 12-17, doi:10.1145/979743.979766.

[A5] S. Reis, A. Metzger, K. Pohl, Integration Testing in Software Product Line

Engineering: A Model-Based Technique, in Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07), Springer Berlin Heidelberg, 2007, pp. 321-335, doi:10.1007/978-3-540-71289-3_25.

[A6] R. Kolb, D. Muthig, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, in: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06), ACM, 2006, pp. 22-27, doi:10.1145/1147249.1147252.

[A7] K. Kim, H. Kim, M. Ahn, M. Seo, Y. Chang, K.C. Kang, ASADAL: A Tool System for Co-development of Software and Test Environment Based on Product Line Engineering, in: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), ACM, 2006, pp. 783-786, doi:10.1145/1134285.1134412.

[A8] T. Kishi, N. Noda, Design Testing for Product Line Development Based on Test Scenarios, in: Presented at Software Product Line Testing Workshop (SPLiT), Boston, MA, 2004.

[A9] T. Kahsai, M. Roggenbach, B.H. Schlingloff, Specifications-Based Testing for Software Product Lines, in: 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), IEEE, 2008, pp. 149-158, doi:10.1109/SEFM.2008.38.

[A10] J. Pesonen, M. Katara, T. Mikkonen, Production-Testing of Embedded Systems with Aspects, in: Proceedings of 1st Haifa International Conference on Hardware and Software Verification and Testing (HVC'05), Springer-Verlag Berlin, Heidelberg, 2006, pp. 90-102, doi:10.1007/11678779_7.

[A11] T. Trew, Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products, in: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05), Springer-Verlag Berlin, Heidelberg, 2005, pp. 137-149, doi:10.1007/11554844_17.

[A12] Y. Ghanam, F. Maurer, Linking Feature Models to Code Artifacts Using

Executable Acceptance Tests, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 211-225.

[A13] J. Camara, A. Kobsa, Facilitating Controlled Tests of Website Design Changes Using Aspect-Oriented Software Development and Software Product Lines, Transactions on Large-Scale Data-and Knowledge-Centered Systems I, Lecture Notes in Computer Science, Vol. 5740, Springer Berlin Heidelberg, 2009, pp. 116-135, doi:10.1007/978-3-642-03722-1_5.

[A14] M. Ardis, N. Daley, D. Hoffman, H. Siy, D. Weiss, Software Product Lines: A Case Study, Software: Practice and Experience, Vol. 30, Issue 7, 2000, pp. 825-847.

[A15] B.J. Geppert, J. Li, F. Rossler, D.M. Weiss, Towards Generating Acceptance Tests for Product Lines, in: 8th International Conference on Software Reuse, Madrid, Spain, 2004.

[A16] C. Denger, R. Kolb, Testing and Inspecting Reusable Product Line Components: First Empirical Results, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06), ACM, 2006, pp. 184-193, doi:10.1145/1159733.1159762.

[A17] K. Pohl, A. Metzger, Software Product Line Testing, Communications of the ACM Software Product Line, Vol. 49, Issue 12, ACM, 2006, pp. 78-81, doi:10.1145/1183236.1183271.

[A18] S. Ajila, R. Dumitrescu, Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution, Journal of Systems and Software, Vol. 80, Issue 1, Elsevier Science, 2007, pp. 74-91, doi:10.1016/j.jss.2006.05.034.

[A19] A. Bertolino, S. Gnesi, PLUTO: A Test Methodology for Product-Families, in: 5th International Workshop Software Product-Family Engineering, Siena, Italy, 2003.

[A20] C. Nebut, F. Fleurey, Y.L. Traon, J.-M. Jezequel, A Requirement-based Approach to Test Product Families, in: International Workshop on Product Family Engineering (PFE), 2003.

[A21] E.M. Olimpiew, H. Gomaa, Reusable Model-based Testing, in: Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering (ICSR'09), Springer-Verlag Berlin, Heidelberg, 2009, pp. 76-85, doi:10.1007/978-3-642-04211-9_8.

[A22] J. Al Dallal, P. Sorenson, Testing Software Assets of Framework-based Product Families During Application Engineering Stage, Journal of Software, Vol. 3, No. 5, Academy Publisher, 2008, pp. 11-25, doi:10.4304/jsw.3.5.11-25.

[A23] A. Reuys, E. Kamsties, K. Pohl, Model-based System Testing of Software Product Families, in: Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05), Springer Verlag Berlin, Heidelberg, 2005, pp. 519-534, doi:10.1007/11431855_36.

[A24] S. Kang, J. Lee, M. Kim, W. Lee, Towards a Formal Framework for Product Line Test Development, in: Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT 2007), IEEE, 2007, pp. 921-926, doi:10.1109/CIT.2007.40.

[A25] B.P. Lamancha, M. Polo, M. Piattini, An Automated Model-Driven Testing Framework for Model-Driven Development and Software Product Lines, in: International Conference on Evaluation of Novel Approaches to Software Engineering, 2010.

[A26] E.D.M. Rodrigues, L.D. Viccari, A.F. Zorzo, PLeTs-Test Automation Using Software Product Lines and Model Based Testing, in: International Conference on Software Engineering and Knowledge Engineering, 2010.

[A27] B.P. Lamancha, M.P. Usaola, Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage, in: Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems

(ICTSS'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 111-125.

[A28] I. Cabral, M.B. Cohen, G. Rothermel, Improving the Testing and Testability of Software Product Lines, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 241-255.

[A29] S. Oster, F. Markert, P. Ritter, Automated Incremental Pairwise Testing of Software Product Lines, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 196-210.

[A30] V. Stricker, A. Metzger, K. Pohl, Avoiding Redundant Testing in Application Engineering, in: J. Bosch, J. Lee (eds.) SPLC 2010. LNCS, Vol. 6287, Springer, Heidelberg, 2010, pp. 226-240.

[A31] G. Perrouin, S. Sen, J. Klein, B. Baudry, Y. le Traon, Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines, in: Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2010, pp. 459-468, doi:10.1109/ICST.2010.43.

[B1] A. Reuys, S. Reis, E. Kamsties, K. Pohl, Derivation of Domain Test Scenarios from Activity Diagrams, in: Proceedings of the International Workshop on Product Line Engineering the Early Steps: Planning, Modeling, and Managing (PLEES'03), 2003.

[B2] C. Nebut, F. Fleurey, Y.L. Traon, J.-M. Jezequel, A Requirement-based Approach to Test Product Families, in: International Workshop on Product Family Engineering (PFE), 2003.

[B3] Y. Feng, X. Liu, J. Kerridge, A Product Line Based Aspect-Oriented Generative Unit Testing Approach to Building Quality Components, in: Proceedings of the 31st Annual international Computer Software and Applications Conference (COMPSAC'07), Vol. 2, IEEE, 2007, pp. 403-408, doi:10.1109/COMPSAC.2007.35.

[B4] S. Reis, A. Metzger, K. Pohl, Integration Testing in Software Product Line Engineering: A Model-Based Technique, in Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07), Springer Berlin Heidelberg, 2007, pp. 321-335, doi:10.1007/978-3-540-71289-3_25.

[B5] A. Reuys, S. Reis, E. Kamsties, K. Pohl, The ScenTED Method for Testing Software Product Lines, in: Software Product Lines, Springer, Heidelberg, 2006, pp. 479-520, doi:10.1007/978-3-540-33253-4_13.

[B6] T. Kishi, N. Noda, Design Testing for Product Line Development Based on Test Scenarios, in: Presented at Software Product Line Testing Workshop (SPLiT), Boston, MA, 2004.

[B7] J.J. Li, D.M. Weiss, J.H. Slye, Automatic Integration Test Generation from Unit Tests of eXVantage Product Family, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'07), Kyoto, Japan, pp. 73-80, 2007.

[B8] S. Reis, A. Metzger, K. Pohl, A Reuse Technique for Performance Testing of Software Product Lines, in: Proceedings of the International Workshop on Software Product Line Testing, Mannheim University of Applied Sciences, Report No. 003.06, 2006, pp. 5-10.

[B9] S. Mishra, Specification Based Software Product Line Testing: A Case Study, in: Proceedings of the Concurrency: Specification and Programming Workshop, 2006, pp. 243-254.

[B10] S. Bashardoust-Tajali, J-P. Corriveau, On Extracting Tests from A Testable Model in the Context of Domain Engineering, in: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), IEEE, 2008, pp. 98-107, doi:10.1109/ICECCS.2008.17.

[B11] J. Hartmann, M. Vieira, A. Ruder, UML-based Approach for Validating Product Lines, in: International Workshop on Software Product Line Testing (SPLiT), Avaya Labs Technical Report, Boston, USA, 2004, pp. 58-64.

[B12] A. Bertolino, S. Gnesi, PLUTO: A Test Methodology for Product-Families, in: 5th International Workshop Software Product-Family Engineering, Siena, Italy, 2003.

[B13] A. Bertolino, S. Gnesi, Use case-based Testing of Product Lines, in: Proceedings of ESEC/FSE, ACM, 2003, pp. 355-358, doi:10.1145/940071.940120.

[B14] E. Kamsties, K. Pohl, S. Reis, A. Reuys, Testing Variabilities in Use Case Models, in: Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE 2003), Springer, Heidelberg, 2003, pp. 6-18, doi:10.1007/978-3-540-24667-1_2.

[B15] C. Nebut, Y.L. Traon, J.-M. Jezequel, System Testing of Product Lines: From Requirements to Test Cases Software Product Lines, Software Product Lines, Springer Berlin Heidelberg, 2006, pp. 447-477, doi:10.1007/978-3-540-33253-4_12.

[B16] A. Reuys, E. Kamsties, K. Pohl, Model-based System Testing of Software Product Families, in: Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05), Springer Verlag Berlin, Heidelberg, 2005, pp. 519-534, doi:10.1007/11431855_36.

[B17] E.M. Olimpiew, H. Gomaa, Model-Based Testing for Applications Derived from Software Product Lines, in: 1st Workshop on Advances in Model-Based Software Testing (A-MOST'05), ACM Press, 2005.

[B18] E.M. Olimpiew, H. Gomaa, Model-Based Test Design for Software Product Lines, in: International Workshop on Software Product Line Testing (SPLiT 2008), 2008.

[B19] J.C. Duenas, J. Mellado, R. Ceron, J.L. Arciniegas, J.L. Ruiz, R. Capilla, Model Driven Testing in Product Family Context, in: First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, 2004.

[B20] S. Kang, J. Lee, M. Kim, W. Lee, Towards a Formal Framework for Product

Line Test Development, in: Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT 2007), IEEE, 2007, pp. 921-926, doi:10.1109/CIT.2007.40.

[B21] S. Weissleder, D. Sokenou, B.H. Schlingloff, Reusing State Machines for Automatic Test Generation in Product Lines, in: 1st Workshop on Model-based Testing in Practice, 2008.

[B22] T. Kahsai, M. Roggenbach, B.H. Schlingloff, Specifications-Based Testing for Software Product Lines, in: 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), IEEE, 2008, pp. 149-158, doi:10.1109/SEFM.2008.38.

[B23] E. Uzuncaova, D. Garcia, S. Khurshid, D. Batory, Testing Software Product Lines using Incremental Test Generation, in: 19th International Symposium on Software Reliability Engineering (ISSRE 2008), IEEE, 2008, pp. 249-258, doi:10.1109/ISSRE.2008.56.

[B24] B.J. Geppert, J. Li, F. Rossler, D.M. Weiss, Towards Generating Acceptance Tests for Product Lines, in: 8th International Conference on Software Reuse, Madrid, Spain, 2004.

[B25] J.D. McGregor, Testing a Software Product Line, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2001-TR-022, 2001.

[B26] R. Kolb, D. Muthig, Challenges in Testing Software Product Lines, in: Proceedings of CONQUEST'03, Nuremberg, Germany, 2003, pp. 81-95.

[B27] R. Kolb, D. Muthig, Techniques and Strategies for Testing Component-based Software and Product Lines, Development of Component-based Information Systems, Advances in Management Information Systems 2 (2006) 123-139, Chapter 7.

[B28] A. Tevanlinna, J. Taina, R. Kauppinen, Product Family Testing: A Survey,

ACM SIGSOFT Software Engineering Notes, Vol.29, Issue 2, ACM, 2004, pp. 12-17, doi:10.1145/979743.979766.

[B29] L. Jin-hua, L. Qiong, L. Jing, The W-Model for Testing Software Product Lines, in: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT'08), Vol. 1, IEEE, 2008, pp. 690-693, doi:10.1109/ISCSCT.2008.34.

[B30] Y. Ghanam, S. Park, F.A. Maurer, A Test-Driven Approach to Establishing and Managing Agile Product Lines, in: The 5th SPLiT Workshop SPLC, Ireland, 2008.

[B31] R. Kolb, A Risk-Driven Approach for Efficiently Testing Software Product Lines, in: IESEF'03 Fraunhofer Institute for Experimental Software Engineering, 2003.

[B32] S. Oster, A. Schurr, I. Weisemoller, Towards Software Product Line Testing Using Story Driven Modeling, in: Proceedings of the 6th International Fujaba Days, 2008, pp. 48-55.

[B33] J. Al Dallal, P. Sorenson, Testing Software Assets of Framework-based Product Families During Application Engineering Stage, Journal of Software, Vol. 3, No. 5, Academy Publisher, 2008, pp. 11-25, doi:10.4304/jsw.3.5.11-25.

[B34] H. Zeng, W. Zhang, D. Rine, Analysis of Testing Effort by Using Core Assets in Software Product Line Testing, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'04), Boston, MA, 2004, pp. 1-6.

[B35] U. Dowie, N. Gellner, S. Hanssen, A. Helferich, G. Herzwurm, S. Schockert, Quality Assurance of Integrated Business Software: An Approach to Testing Software Product Lines, in: Proceedings of the 13th European Conference on Information Systems, Information Systems in a Rapidly Changing Economy (ECIS), 2005.

[B36] J.D. McGregor, Toward a Fault Model for Software Product Lines, in: Proceedings of Fifth International Workshop on Software Product Line Testing (SPLiT'08), 2008, p. 27.

[B37] T. Trew, What Design Policies Must Testers Demand From Product Line Architects?, in: Proceedings of International Workshop on Software Product Line Testing, 2004.

[B38] R. Kolb, D. Muthig, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, in: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06), ACM, 2006, pp. 22-27, doi:10.1145/1147249.1147252.

[B39] J.D. McGregor, P. Sodhani, S. Madhavapeddi, Testing Variability in a Software Product Line, in: Proceedings of the International Workshop on Software Product Line Testing, Avaya Labs, ALR-2004-031, 2004, pp. 45-50.

[B40] C. Condron, A Domain Approach to Test Automation of Product Lines, in: International Workshop on Software Product Line Testing, 2004.

[B41] P. Knauber, J. Schneider, Tracing Variability from Implementation to Test Using Aspect-Oriented Programming, in: International Workshop on Software Product Line Testing (SPLiT), 2004.

[B42] J.J. Williams, Test Case Management of Controls Product Line Points of Variability, in: International Workshop on Software Product Line Testing (SPLiT), 2004.

[B43] K.D. Scheidemann, Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems, in: Proceedings of the 10th International Software Product Line Conference (SPLC'06), IEEE, 2006, pp. 75-84.

[B44] T. Gustafsson, An Approach for Selecting Software Product Line Instances for Testing, in: International Workshop on Software Product Line Testing, 2007.

[B45] M.B. Cohen, M.B. Dwyer, J. Shi, Coverage and Adequacy in Software Product Line Testing, in: Proceedings of the ISSTA 2006 Workshop on Software Architecture for Testing and Analysis (ROSATEA'06), ACM, 2006, pp. 53-63, doi:10.1145/1147249.1147257.

[B46] R. Kauppinen, J. Taina, A. Tevanlinna, Hook and Template Coverage Criteria for Testing Framework-based Software Product Families, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'04), Boston, MA, USA, 2004, pp. 7-12.

[B47] M. Jaring, R.L. Krikhaar, J. Bosch, Modeling Variability and Testability Interaction in Software Product Line Engineering, in: Proceedings of 7th International Conference on Composition-based Software Systems (ICCBSS 2008), IEEE, 2008, pp. 120-129, doi:10.1109/ICCBSS.2008.9.

[C1] I. Schaefer, L. Bettini, F. Damiani, N. Tanzarella, Delta-Oriented Programming of Software Product Lines, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 77-91.

[C2] M. Lochau, I. Schaefer, J. Kamischke, S. Lity, Incremental Model-Based Testing of Delta-Oriented Software Product Lines, in: Proceedings of the 6th International Conference on Tests and Proofs (TAP'12), Springer-Verlag Berlin, Heidelberg, 2012, pp. 67-82, doi:10.1007/978-3-642-30473-6_7.

[C3] M. Dukaczewski, I. Schaefer, R. Lachmann, M. Lochau, Requirements-Based Delta-Oriented SPL Testing, in: Proceedings of 4th International Workshop on Product Line Approaches in Software Engineering (PLEASE), IEEE, 2013, pp. 49-52, doi:10.1109/PLEASE.2013.6608665.

[C4] L. Linsbauer, E.R. Lopez-Herrejon, A. Egyed, Recovering Traceability Between Features and Code in Product Variants, in: Proceedings of the 17th International Software Product Line Conference (SPLC'13), ACM, 2013, pp. 131-140, doi:10.1145/2491627.2491630.

[C5] Y. Ghanam, F. Maurer, Linking Feature Models to Code Artifacts Using Executable Acceptance Tests, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Springer-Verlag Berlin, Heidelberg, 2010, pp. 211-225.

[C6] S. Wang, A. Gotlieb, S. Ali, M. Liaaen, in: Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science, Vol. 8107, Springer Berlin Heidelberg, 2013, pp. 237-253, doi:10.1007/978-3-642-41533-3_15.

[C7] C.H.P. Kim, S. Khurshid, D. Batory, Shared Execution for Efficiently Testing Product Lines, in: Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2012, pp. 221-230, doi:10.1109/ISSRE.2012.23.

[C8] C.H.P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, M. D'Amorim, SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), ACM, 2013, pp. 257-267, doi:10.1145/2491411.2491459.

[D1] J.J. Li, D.M. Weiss, J.H. Slye, Automatic Integration Test Generation from Unit Tests of eXVantage Product Family, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'07), Kyoto, Japan, pp. 73-80, 2007.

[D2] E. Olimpiew, H. Gomaa, Reusable System Tests for Applications Derived from Software Product Lines, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'05), Rennes, France, 2005a.

[D3] A. Reuys, S. Reis, E. Kamsties, K. Pohl, The ScenTED Method for Testing Software Product Lines, in: Software Product Lines, Springer, Heidelberg, 2006, pp. 479-520, doi:10.1007/978-3-540-33253-4_13.

[D4] A. Wubbeke, Towards an Efficient Reuse of Test Cases for Software Product Lines, in: Proceedings of Software Product Line Conference (SPLC'08), 2008, pp.

361-368.

[D5] J.D. McGregor, Testing a Software Product Line, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2001-TR-022, 2001.

[D6] R. Kauppinen, J. Taina, RITA Environment for Testing Framework-based Software Product Lines, in: SPLST'03 Proceedings of the 8th Symposium on Programming Languages and Software Tools, Kuopio, Finland, 2003, pp. 58-69.
[D7] R. Kolb, A Risk-Driven Approach for Efficiently Testing Software Product Lines, in: IESEF'03 Fraunhofer Institute for Experimental Software Engineering, 2003.

[D8] L. Jin-hua, L. Qiong, L. Jing, The W-Model for Testing Software Product Lines, in: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT'08), Vol. 1, IEEE, 2008, pp. 690-693, doi:10.1109/ISCSCT.2008.34.

[D9] S. Reis, A.P.K. Metzger, A Reuse Technique for Performance Testing of Software Product Lines, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'05), Baltimore, Maryland, USA, 2006.

[D10] Y. Feng, X. Liu, J. Kerridge, A Product Line Based Aspect-Oriented Generative Unit Testing Approach to Building Quality Components, in: Proceedings of the 31st Annual international Computer Software and Applications Conference (COMPSAC'07), Vol. 2, IEEE, 2007, pp. 403-408, doi:10.1109/COMPSAC.2007.35.

[D11] D. Ganesan, U. Maurer, M. Ochs, B. Snoek, M. Verlage, Towards Testing Response Time of Instances of a Web-based Product Line, in: Proceedings of International Workshop on Software Product Line Testing (SPLiT'05), Rennes, France, 2005.

[D12] M. Jaring, R.L. Krikhaar, J. Bosch, Modeling Variability and Testability

Interaction in Software Product Line Engineering, in: Proceedings of 7th International Conference on Composition-based Software Systems (ICCBSS 2008), IEEE, 2008, pp. 120-129, doi:10.1109/ICCBSS.2008.9.

[D13] R. Kauppinen, Testing Framework-Based Software Product Lines, Master's Thesis, University of Helsinki Department of Computer Science, 2003.

[D14] O.O. Edwin, Testing in Software Product Lines, Master's Thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden, 2007.

[D15] A. Tevanlinna, J. Taina, R. Kauppinen, Product Family Testing: A Survey, ACM SIGSOFT Software Engineering Notes, Vol.29, Issue 2, ACM, 2004, pp. 12-17, doi:10.1145/979743.979766.

[D16] H. Zeng, W. Zhang, D. Rine, Analysis of Testing Effort by Using Core Assets in Software Product Line Testing, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'04), Boston, MA, 2004, pp. 1-6.

[D17] J.J. Li, B. Geppert, F. Roessler, D. Weiss, Reuse Execution Traces to Reduce Testing of Product Lines, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'07), Kyoto, Japan, 2007.

[D18] J.D. McGregor, Structuring Test Assets in a Product Line Effort, in: Proceedings of the 2nd International Workshop on Software Product Lines: Economics, Architectures, and Implications (ICSE'01), Toronto, Ontario, Canada, 2001, pp. 89-92.

[D19] J.D. McGregor, Building Reusable Test Assets for a Product Line, in: Proceedings of 7th International Conference on Software Reuse (ICSR'02), Austin, Texas, USA, 2002, pp. 345-346.

[D20] C. Nebut, Y.L. Traon, J.-M. Jezequel, System Testing of Product Lines: From Requirements to Test Cases Software Product Lines, Software Product

Lines, Springer Berlin Heidelberg, 2006, pp. 447-477, doi:10.1007/978-3-540-33253-4_12.

[D21] E.M. Olimpiew, H. Gomaa, Reusable Model-based Testing, in: Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering (ICSR'09), Springer-Verlag Berlin, Heidelberg, 2009, pp. 76-85, doi:10.1007/978-3-642-04211-9_8.

[D22] M.B. Cohen, M.B. Dwyer, J. Shi, Coverage and Adequacy in Software Product Line Testing, in: Proceedings of the ISSTA 2006 Workshop on Software Architecture for Testing and Analysis (ROSATEA'06), ACM, 2006, pp. 53-63, doi:10.1145/1147249.1147257.

[D23] R. Kauppinen, J. Taina, A. Tevanlinna, Hook and Template Coverage Criteria for Testing Framework-based Software Product Families, in: Proceedings of the International Workshop on Software Product Line Testing (SPLiT'04), Boston, MA, USA, 2004, pp. 7-12.

[D24] R. Kolb, D. Muthig, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, in: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06), ACM, 2006, pp. 22-27, doi:10.1145/1147249.1147252.

[D25] J.D. McGregor, P. Sodhani, S. Madhavapeddi, Testing Variability in a Software Product Line, in: Proceedings of the International Workshop on Software Product Line Testing, Avaya Labs, ALR-2004-031, 2004, pp. 45-50.

[D26] T. Kishi, N. Noda, Formal Verification and Software Product Lines, in: Communications of the ACM Software Product Line, Vol. 49, Issue 12, ACM, 2006, pp. 73-77, doi:10.1145/1183236.1183270.

Appendix G

Generated XML Based Reflexion Model

Decomposition Viewpoint - First Model

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<Reflexion_Model_Decomposition_Style>

<Convergence>

<Common_Convergence>

<Element>

<Name>Subsystem B</Name>

<Sub_Elements>

<Name>Module B4</Name>

<Name>Module B1</Name>

<Name>Module B2</Name>

<Name>Module B3</Name>

</Sub_Elements>

</Element>

<Element>

<Name>Module B4</Name>

<Parent_Element>Subsystem B</Parent_Element>

</Element>

<Element>

<Name>Subsystem C</Name>

<Sub_Elements>

<Name>Module C4</Name>

<Name>Module C1</Name>

<Name>Module C2</Name>

<Name>Module C3</Name>

</Sub_Elements>

</Element>

<Element>

<Name>Module C4</Name>

<Parent_Element>Subsystem C</Parent_Element>

</Element>

<Element>

<Name>Subsystem D</Name>

<Sub_Elements>

<Name>Module D4</Name>

<Name>Module D1</Name>

<Name>Module D2</Name>

<Name>Module D3</Name>

</Sub_Elements>

</Element>

<Element>

<Name>Module D4</Name>

<Parent_Element>Subsystem D</Parent_Element>

</Element>

</Common_Convergence>

<Variant_Convergence>

<Element>

<Name>Module B1</Name>

<Parent_Element>Subsystem B</Parent_Element>

</Element>

</Variant_Convergence>

</Convergence>

<Absence>

<Common_Absence>

<Element>

<Name>Module A</Name>

</Element>

<Element>

<Name>Module F</Name>

</Element>

</Common_Absence>

<Variant_Absence>

<Element>

<Name>Module C1</Name>

<Parent_Element>Subsystem C</Parent_Element>

<Absence_Reason>MIN/MAX value constraint of the alternative relation is violated</Absence_Reason>

</Element>

<Element>

<Name>Module C2</Name>

<Parent_Element>Subsystem C</Parent_Element>

<Absence_Reason>MIN/MAX value constraint of the alternative relation is violated</Absence_Reason>

</Element>

<Element>

<Name>Module C3</Name>

<Parent_Element>Subsystem C</Parent_Element>

<Absence_Reason>MIN/MAX value constraint of the alternative relation is violated</Absence_Reason>

</Element>

<Element>

<Name>Module E</Name>

</Element>

</Variant_Absence>

</Absence>

<Not_Selected_Variants>

<Element>

<Name>Module B2</Name>

<Parent_Element>Subsystem B</Parent_Element>

</Element>

<Element>

<Name>Module B3</Name>

<Parent_Element>Subsystem B</Parent_Element>

</Element>

<Element>

<Name>Module D1</Name>

<Sub_Elements>

<Name>Module D11</Name>

</Sub_Elements>

<Parent_Element>Subsystem D</Parent_Element>

</Element>

<Element>

<Name>Module D2</Name>

<Parent_Element>Subsystem D</Parent_Element>

</Element>

<Element>

<Name>Module D3</Name>

<Parent_Element>Subsystem D</Parent_Element>

</Element>

<Element>

<Name>Module D11</Name>

<Parent_Element>Module D1</Parent_Element>

</Element>

</Not_Selected_Variants>

<Statistical_Data>

<Total_Number_Of_Element_In_The_Reference_Architecture>19

</Total_Number_Of_Element_In_The_Reference_Architecture>

<Total_Number_Of_Element_In_The_Application_Architecture>11

</Total_Number_Of_Element_In_The_Application_Architecture>

<Total_Number_Of_Convergence_Element convergence_percentage="53.85" total_number="7">

<Total_Number_Of_Common_Convergence_Element common_convergence_percentage="75">6

</Total_Number_Of_Common_Convergence_Element>

<Total_Number_Of_Variant_Convergence_Element variant_convergence_percentage="20">1

</Total_Number_Of_Variant_Convergence_Element>

</Total_Number_Of_Convergence_Element>

<Total_Number_Of_Absence_Element absence_percentage="46.15" total_number="6">

<Total_Number_Of_Common_Absence_Element common_absence_percentage="25">2

</Total_Number_Of_Common_Absence_Element>

<Total_Number_Of_Variant_Absence_Element variant_absence_percentage="80">4

</Total_Number_Of_Variant_Absence_Element>

</Total_Number_Of_Absence_Element>

<Total_Number_Of_Delta_Element>0</Total_Number_Of_Delta_Element>

<Total_Number_Of_Not_Selected_Variant_Element not_selected_variant_percentage="54.55">6

</Total_Number_Of_Not_Selected_Variant_Element>

</Statistical_Data>

</Reflexion_Model_Decomposition_Style>