MODEL-DRIVEN ENGINEERING OF SOFTWARE ARCHITECTURE VIEWPOINTS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Elif Demirli
September, 2012

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

Asst. Prof. Dr. Bedir Tekinerdoğan (Supervisor)

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

Prof. Dr. Özgür Ulusoy

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

Asst. Prof. Dr. Kayhan İmre

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural

Director of the Graduate School

ABSTRACT

MODEL-DRIVEN ENGINEERING OF SOFTWARE ARCHITECTURE VIEWPOINTS

Elif Demirli
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. Bedir Tekinerdoğan
September, 2012

A common practice in software architecture design is to apply so-called architectural views to design software architecture for the various stakeholder concerns. Architectural views are usually developed based on architectural viewpoints which define the conventions for constructing, interpreting and analyzing views. So far most architectural viewpoints seem to have been primarily used either to support the communication among stakeholders, or at the best to provide a blueprint for the detailed design.

In this thesis, we provide a software language engineering approach to define viewpoints as domain specific languages. This enhances the formal precision of architectural viewpoints and leads to executable views that can be interpreted and analyzed by tools. We illustrate our approach for defining domain specific languages for the viewpoints of the Views and Beyond framework. The approach is implemented as an Eclipse plug-in, SAVE-Bench tool, which can be used to define different views based on the predefined software architecture viewpoints. The tool also supports automatic generation of architecture documentation from view models.

Keywords: Software Architecture Viewpoints, Software Language Engineering, Domain-Specific Modeling, Model-Driven Engineering, Tool Support.

ÖZET

YAZILIM MİMARİSİ BAKIŞ AÇILARI İÇİN MODEL GÜDÜMLÜ MÜHENDİSLİK

Elif Demirli

Bilgisayar Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Yrd. Doç. Dr. Bedir Tekinerdoğan

Eylül, 2012

Yazılım mimarisi tasarımında yaygın pratiklerden biri yazılım mimarisini çeşitli paydaş ilgilerine yönelik tasarlayabilmek için mimari görünümlerini kullanmaktır. Mimari görünümleri genellikle bu görünümleri oluşturmayı, yorumlamayı ve analiz etmeyi sağlayan kuralları tanımlayan mimari bakış açılarını temel alarak geliştirilir. Şimdiye kadar çoğu mimari bakış açısının esasen paydaşlar arasındaki iletişimi desteklemek ya da en iyi ihtimalle detaylı tasarım için bir plan sağlamak amacıyla kullanıldığı görülmektedir.

Bu tezde mimari bakış açılarını alana özgü dil olarak tanımlamak için bir yazılım dil mühendisliği yaklaşımı sunuyoruz. Bu, mimari bakış açılarının formalliğini iyileştirirken bir yandan da araçlar tarafından yorumlanıp analiz edilebilen çalıştırılabilir görünüm modellerine öncülük ediyor. Mimari bakış açılarını alana özgü dil olarak tanımlama çalışmamızı Görünümler ve Ötesi yaklaşımı için gösterdik. Yaklaşımımız çeşitli görünümleri modellemeyi destekleyen Eclipse eklentisi SAVE-Bench yazılım aracı olarak geliştirildi. Araç aynı zamanda görünüm modellerinden otomatik mimari dökümantasyonu üretmeyi de destekliyor.

Keywords: Yazılım Mimari Bakış Açıları, Yazılım Dil Mühendisliği, Alana-Özgü Modelleme, Model Güdümlü Mühendislik, Araç Desteği.

Acknowledgement

I would like to express my sincere gratitude to my supervisor Asst. Prof. Dr. Bedir Tekinerdoğan for his invaluable guidance, support and understanding during this thesis. He encouraged and motivated me during my whole research.

I am thankful to Prof. Dr. Özgür Ulusoy and Asst. Prof. Dr. Kayhan İmre for kindly accepting to be in the committee and also for giving their precious time to read and review this thesis.

I am grateful to The Scientific and Technological Research Council of Turkey (TÜBİTAK-BİDEB) for the financial support they provided during the time span of this thesis.

I would like to thank to my housemate Özlem, my friends Seher, Şadiye, Püren and Başak for their valuable friendship and the enjoyable time we spent together.

Last but not least, I would like to thank my family, my mother Makbule, my father Ekrem and my sister Esra for being in my life, for their endless, unconditional love and support to me. With very special thanks, I dedicate this thesis to them.

Contents

List of Figures

Chapter 1

Introduction

1.1. Software Architecture Design

As the size and complexity of software systems increases, software architecture has emerged as an important sub-discipline of software engineering. A *software architecture* for a program or computing system consists of the structure or structures of that system, which comprise elements, the externally visible properties of those elements, and the relationships among them [6]. Since it depicts the high-level structure of the system, software architecture is a valuable artifact for both communicating and designing the system. Representing a common abstraction of a system, software architecture forms a basis for understanding and communication among stakeholders who have various concerns in the construction of the software system. In addition to this, as one of the earliest artifact of the software development life cycle, software architecture embodies early design decisions, which impacts the system's detailed design, implementation, deployment and maintenance. That is why; it must be carefully documented and analyzed.

Software architecture is not a single one-dimensional structure but it consists of a set of structures. This can be better explained via house architecture analogy. In order to build a house and reason about its architecture, every stakeholder of the house either uses or creates a particular plan that satisfies his own interests. The house architect designs the skeleton of the house. Interior designer defines the interior architecture plan and the electrician sets up wiring plan based on the house architect's plan. All of these plans are different entities and every stakeholder is interested in a few of them. Individually, none of those plans can be called as house architecture. However, when all of them are brought together, they constitute the architecture of the house. The same situation applies in software development, too. A software system has a set of stakeholders who have special interests on the overall system. A software developer is interested in how the system is structured as a set of implementation units, a performance engineer is interested in the organization of the run-time elements and a project manager is concerned about the distribution of implementation units among development teams. Each of these structures is a part of the software architecture, however; none of them can be called as software architecture by itself. In order to enable dealing with those different structures easily architectural view concept was introduced.

An *architectural view* is a representation of a set of system elements and relations associated with them to support a particular concern [6]. Each view shows the system from a different point of view. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. The conventions for constructing and using a view are specified by *viewpoint*s [6]. Viewpoints basically define element and relation types that can be used for the corresponding view, together with some set of constraints on their use. An *architectural framework* organizes and structures the proposed architectural viewpoints. Different architectural frameworks have been proposed in the literature. Examples of architectural frameworks include the Kruchten's 4+1 view model [24], the Siemens Four View Model [17], and the Views and Beyond approach (V&B) [6][7].

1.2. Problem Statement

Obviously the notion of architectural view plays an important role in modeling and documenting architectures. Architectural views are intended to be used for communication, design and analysis. The quality, expressiveness and value of view models are based on the corresponding viewpoint definitions. When examined, it is observed that so far most architectural viewpoints seem to have been primarily used either to support the communication among stakeholders, or at the best to provide a blueprint for the detailed design. These viewpoints were defined in a notation and representation neutral manner to increase their use and flexibility. That is why; most viewpoint definitions are high-level and abstract. A comprehensive analysis and design process is required to develop tools for modeling viewpoint specifications. Consequently, the derivation of formal view models and performing formal analysis of the specifications produced becomes harder.

The lack of a formal approach for defining viewpoints results in less precise viewpoint definitions. From a historical perspective it can be observed that viewpoints defined later are more precise and consistent than the earlier approaches but a close analysis shows that even existing viewpoints lack some precision. Moreover, since existing frameworks provide mechanisms to add new viewpoints the risk of introducing imprecise viewpoints is high. The development of a proper and effective architecture is highly dependent on the corresponding documentation. An incomplete or imprecise viewpoint will impede the understanding and application of the viewpoints to derive the corresponding architectural views, and likewise lower the quality of the architectural document. The resulting view models lead ambiguous interpretations.

When the function of architectural views in software development lifecycles is examined, we observe that architectural views are not single isolated artifacts. They have relations both among themselves and to other software development artifacts. In order to enable consistency and automation among those artifacts, executable view models are required. When current view modeling practices are analyzed, we observe that view models are not at the desired level of formality. The reason behind it is the lack of formal viewpoint definitions on top of which formal and automatically processable executable view models can be defined.

In addition to these, since the language to define views, namely viewpoints, are not formally defined, developing a tool support for view modeling requires heavyweight analysis step to lift viewpoint definitions up to proper level of formality. In the literature, there is not much architecture modeling tools that puts architectural views at the center. This lack leads view modelers to simple box-and-line diagrams which is not a healthy way since the ambiguities in notation can cause misinterpretations.

In summary, we have identified the following problems in current architecture view modeling practices:

- Lack of formal approach for defining viewpoints
- Imprecise and vague viewpoint definitions in the literature
- Lack of tool support

1.3. Approach

In order to address above problems, we propose that viewpoints should be also defined as formal languages. We recognize that viewpoints are in fact metamodels and we provide a Model Driven Engineering (MDE) approach for defining viewpoints as domain specific languages (DSLs). This enhances the formal precision of architectural viewpoints and likewise helps to share the additional benefits of domain specific languages, i.e. defining executable views.

First, we identified that viewpoints are in fact domain specific languages. A domain specific language is an executable specification language that offers proper abstractions and notations for expressing a particular problem domain [8]. Viewpoints are DSLs since they provide particular abstractions and notations for specific stakeholder concerns. We selected Views and Beyond architecture framework to show our process of defining DSLs for viewpoints. We recognize viewpoints as DSLs that is why we develop metamodel for each viewpoint. In both software language engineering and model-driven development domain, a metamodel is defined as follows:

1) *Definition of abstract syntax:* Abstract syntax describes the vocabulary of concepts provided by the language and how they may be combined to create models. In order to define abstract syntax for viewpoint DSLs, we first needed to identify language constructs that will be used in modeling views. We analyzed the current viewpoint definitions to define the vocabulary of our DSLs. We observe that most viewpoint definitions in Views and Beyond Framework are not at the desired level of formality to map them easily to language. We filled out the missing parts and resolved inconsistencies in language constructs when required. After deciding the language constructs and their interrelations, we define grammar for the viewpoint using Eclipse Xtext tool [37]. Grammar encapsulates the abstract syntax in itself.

2) *Definition of concrete syntax:* Concrete syntax defines the notation that facilitates presentation and construction of models. We define both textual and visual concrete syntax for viewpoints. Textual concrete syntax is embedded in grammar definition. We use Eclipse GMF tools [16] and other supplementary Eclipse plug-ins to define visual concrete syntax. In our analysis on current viewpoint definitions, we observe that usually formal notations or modeling tools are not provided. We make use of the informal and semi-formal notations provided and define our own notation on top of them.

3) *Definition of static semantics:* Static semantics, namely well-formedness rules, provide definition of additional constraint rules on abstract syntax that are hard or impossible to express in standard syntactic formalism of the abstract syntax. In viewpoint definitions of Views and Beyond approach, these constraints are listed as topology constraints. We analyzed them and observed that they are mostly incomplete constraints in natural language. We lift them up to executable constraints embedding them into our DSL definitions as validation codes written in Java.

4) *Definition of semantics:* Semantics is the description of the meaning of the concepts in the abstract syntax. In this work, we don't provide formal semantic specifications. We keep it out of the scope of this thesis.

We followed these steps to define metamodels for the selected viewpoints. We illustrate our approach in a generic way through the thesis such that it can be applied on other viewpoint frameworks. We used various Eclipse tools in our work and present our formal viewpoint specifications as Eclipse plug-ins in our tool SAVE-Bench. SAVE-Bench consists of domain specific languages for Views and Beyond framework viewpoints and it enables modeling textual and visual views that conforms to those viewpoints. It also supports automatic architecture documentation from view models.

1.4. Contribution

The contributions of this thesis can be defined as follows:

- *Systematic approach for modeling architectural viewpoints as DSLs and executable models for the Views and Beyond approach*

We recognized the lack of a formal approach for defining architectural viewpoints. The key premise of this work is recognizing that viewpoints are in fact domain specific languages. We present our software language engineering approach explicitly through the thesis. We analyzed the V&B framework viewpoints and concluded that the viewpoint definitions are not at the desired level of formality to support executable view models. We enhanced those viewpoint definitions and present them in SAVE-Bench tool in order to enable modeling executable views.

- *Evaluation framework for characterizing viewpoint approaches*

In order to evaluate the formality of the viewpoint definitions, we set up an evaluation framework. The framework evaluates the viewpoint definitions from software language engineering perspective with respect to their completeness and degree of formality. We provide evaluation results for V&B viewpoint definitions and observed that viewpoints defined earlier are less precise than those defined later. We also see how our viewpoint DSLs lift the formality and precision of viewpoint definitions up.

- *SAVE-Bench Eclipse Plug-in tool for modeling software architecture viewpoints*

We collected our formal viewpoint definitions for V&B in SAVE-Bench tool. The tool enables defining executable view models using DSLs for viewpoints. It is extensible such that new viewpoint frameworks can be added or new viewpoints can be defined as DSLs into existing frameworks. The tool also supports automatic architecture document generation from view models.

1.5. Outline of the Thesis

This thesis is organized as follows: Chapter 2 provides background information for software architecture views and presents the widely used viewpoint frameworks. Chapter 3 provides an overview of model-driven development. In Chapter 4, first, the idea that viewpoints are in fact domain specific languages is introduced. Case description is provided which will be used as example for modeling views. Then, the domain specific languages for the viewpoints defined in Views and Beyond framework are provided and evaluated with respect to a viewpoint evaluation framework. Chapter 5 presents the SAVE-Bench tool that we have developed for modeling viewpoints and based on these the views. In Chapter 6, automatic architecture documentation from architectural view models is explained. Chapter 7 gives the related work. Finally, Chapter 8 presents the conclusions and discussions.

Chapter 2

Software Architecture Views

Software architecture for a computing system is the *structure* or *structures* of that system, which consists of elements, their externally visible properties and relationships among them [6]. As this definition implies, software architecture is not a single structure, but it consists of lots of overlaying structures. In order to ease dealing with those structures separately, *architectural view* concept was introduced. In this section, we will present the background information on architectural views. Then, we will introduce some selected software architecture and enterprise architecture frameworks that enables modeling the architecture using views.

2.1. Background

Architectural drivers define the concerns of the stakeholders which shape the architecture. A stakeholder is defined as an individual, team, or organization with interests in, or concerns relative to, a system. Each of the stakeholders' concerns impacts the early design decisions that the architect makes [6][20]. A common practice is to model and document different architectural views for describing the architecture according to the stakeholders' concerns. An *architectural view* is a representation of a set of system elements and relations associated with them to support a particular concern [7]. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders.

Architectural views are defined based on viewpoints. An *architectural viewpoint* is a specialization of element and relation types together with a set of constraints on how they can be used [7]. The view and viewpoint concepts are directly addressed in IEEE 1471 standard [20]. Viewpoints encapsulate some design knowledge that addresses a set of stakeholders' concerns. They are independent of systems. When a viewpoint is bound to a system, the resulting model is architectural view of the system. The conceptual model from IEEE 1471 standard describing architectural view and viewpoint concepts are given in Figure 2.1 [20]. As shown in the figure, each architectural view addresses some stakeholders concerns and these concerns also directly affect the viewpoint definitions. Viewpoint definitions are important assets here since they differentiate architectural views to address different concerns.

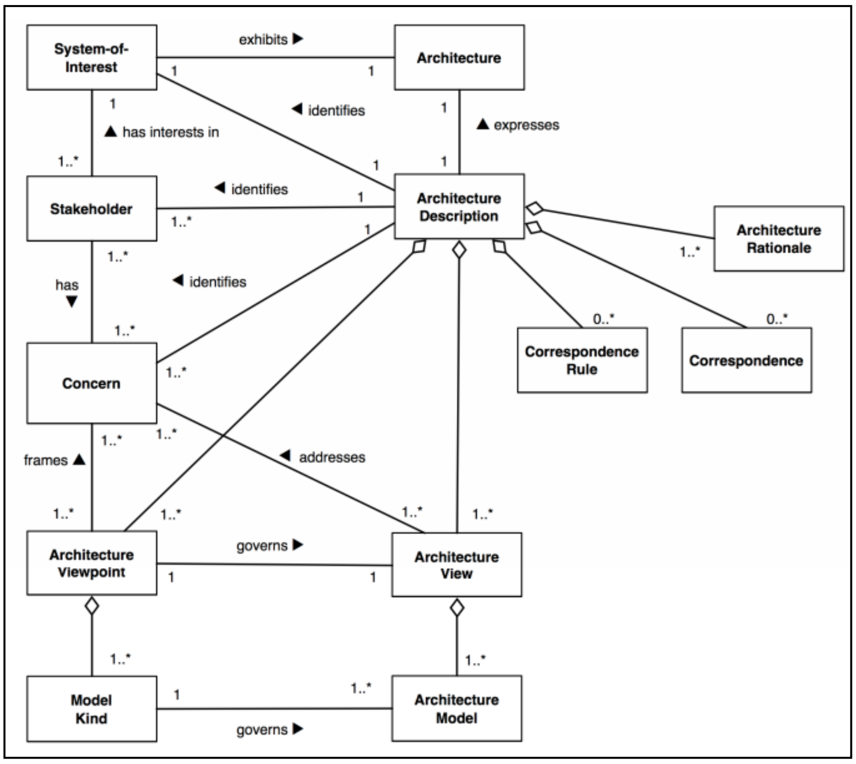


Figure 2.1. IEEE conceptual model for architecture description

A *viewpoint framework* collects and organizes a set of viewpoints to guide the architect [20]. Initially, viewpoint frameworks were introduced as a collection of fixed set of viewpoints to document the architecture. For example, the Rational's Unified Process [], which is based on Kruchten's 4+1 view approach [24] utilizes the logical view, development view, process view and physical view. Lately, this situation has changed. Because of the different concerns that need to be addressed for different systems, the current trend recognizes that the set of views should not be fixed but multiple viewpoints might be introduced instead. For this reason, the IEEE 1471 standard [20] does not commit to any view although it takes a multi-view approach for architectural description.

2.2. Software Architecture Frameworks

Kruchten's 4+1 Viewpoint Framework

Philippe Kruchten's 4+1 set which forms a basis for Rational's Unified Process [] can be seen as the first formal software architecture viewpoint framework in the literature. It describes five different viewpoints to model software architectures.

Figure 2.2 shows the views of Kruchten's framework. *Logical view* can be seen as a kind of object model of the architecture. It is used to support the concerns related to functional requirements. *Process View* takes into account the non-functional requirements such as performance and availability. It captures the concurrency and synchronization aspects of the design. *Physical view* describes the environment in which software executes and shows the mappings of software onto the hardware. *Development view* presents the static organization of the software in its development environment. According to Kruchten, architecture can be organized around these four views. However, a supplementary view (i.e. scenarios) is required to complete the architectural description. This final +1 view serves as glue among other views that ensures the elements of other views work together in harmony.

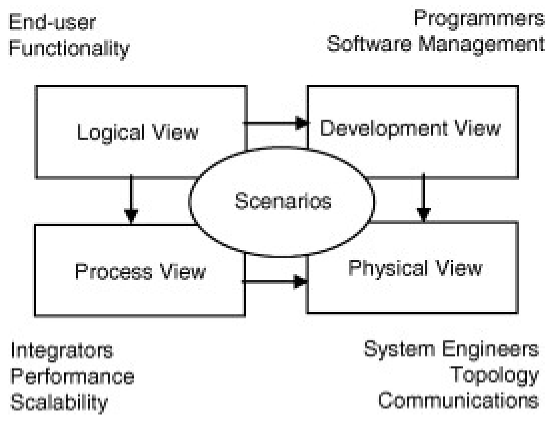


Figure 2.2 Kruchten's 4+1 viewpoint framework

Siemens Four View Framework

Siemens four view framework is a result of a study into the industrial practices of software architecture [17]. The authors found that the structures used to design and document software architecture fall into four broad categories, which they call *conceptual*, *module*, *execution* and *code* structures. Each category addresses different stakeholder concerns.

The views in Siemens Four View framework are not single, isolated models, but several important mappings of structures are explicitly defined in the design approach. The elements of conceptual view are "implemented-by" module view structures, and also "assigned-to" execution view structures. Module view elements can be "located-in" or "implemented-by" code view elements. Code structures can configure execution structures. In other words, there is a strict relation between different views of Siemens Four View framework. Changing the structure or definition of a view will most probably require updating another view.

Rozanski and Woods Framework

Rozanski and Woods [34] address the architecture of large information systems and propose six core viewpoints: *Functional*, *Information*, *Concurrency*, *Development*, *Deployment* and *Operational*.

Functional viewpoint describes the system's runtime functional elements and their responsibilities, interfaces and primary interactions. Its main concerns are functional capabilities and internal structure of the system. As the name implies, *information viewpoint* mainly concerns the information structure of the system. It is used to describe the way that architecture stores, manipulates and distributes information. *Concurrency viewpoint* is used to address the concurrency structure of the system. It shows how functional elements are mapped on concurrency units such as threads and processes in order to clearly identify the parts that can execute concurrently. *Development Viewpoint* addresses software developers' and testers' concerns such as module organization, codeline organization, standardization of design and testing. *Deployment Viewpoint* is used to describe the runtime environment into which the system will be deployed. Finally, *operational viewpoint* describes how the system will be operated, administrated and supported when it is running in its production environment.

Views and Beyond Framework

Views and Beyond framework[6][7] is an open-ended viewpoint framework. Being open-ended framework means that the framework does not limit the number of viewpoints that are defined, any new viewpoints can be introduced. They do not use the term viewpoint explicitly; they refer to it as *style*. A style definition provides the elements and relation types to be used when defining views together with some topological constraints. In V&B framework, there is no limit on the number of styles that can be defined. There is a set of predefined styles that are organized around three main types of architectural styles: Module styles, component-and-connector styles and allocation styles. *Module styles* are used to show how the system is structured as a set of implementation units. Decomposition style is an example to module styles which shows the structure of modules and submodules. *Component and connector styles* are used to show how the system is structures as a set of runtime elements. Pipe-and-filter style is an example to this which shows the data flow between so-called filters that manipulate the data. *Allocation styles* are used to show how the software elements are mapped to non-software elements in its environment. Deployment style is an example to allocation styles and it is used to show how the software elements are mapped on hardware elements and their run-time behavior.

2.3. Enterprise Architecture Frameworks

RM-ODP

Reference Model of Open Distributed Processing (RM-ODP) is a reference model which provides a framework for the standardization of open distributed processing [19]. It provides an enterprise architecture framework which comprises five generic and complementary viewpoints on the system and its environment. The *enterprise viewpoint* focuses on the purpose, scope and policies for the system. It describes the business requirements and how to meet them. *Information viewpoint* focuses on the semantics of the information and information processing performed. It describes the information managed by the system. The *computational viewpoint* enables distribution through functional decomposition on the system into objects which interacts at interfaces. It describes the functionality provided by the system and its functional decomposition. The *engineering viewpoint* focuses on the mechanism and functions required to support distributed interactions between objects in the system. *Technology viewpoint* describes the technologies chosen to provide the processing, functionality and presentation of the information.

Each viewpoint is explicitly specified by a language that defines concepts and rules for specifying ODP systems from the corresponding viewpoints. In addition to this, a UML profile is provided for each viewpoint language.

Zachman's Framework

The Zachman Framework is an enterprise architecture framework which provides a formal and highly structured way of viewing and defining an enterprise [39]. The basic idea behind Zachman's framework is that an item can be described using different ways for different purposes. The framework consists of a two dimensional matrix based on the intersection of six questions and six particular perspectives namely views.

The rows of the framework are as following: Row 1 describes the scope of the system. It is the *planner's view* of the architecture which is an executive summary for planner and investor who wants to estimate the cost and scope of the system. Row 2 is *owner's view* which corresponds to enterprise models that shows business entities, processes and their interrelationships. Row 3 is *designer's view* which shows data elements, logical process flows and functions. Row 4 is *builder's view* which is a more specific version of designer's view. The elements of designer's view are bound to supporting technology for example the programming language that is used. Row 5 is *subcontractor view* which is a detailed specification of the system that is given to programmers who implement individual modules without knowing the overall structure of the system.

Each row in the framework can be described in 6 different representations: data description (what), function description (how), network description (where), people description (who), time description (when), motivation description (why).

TOGAF

The Open Group Architecture Framework is a framework for enterprise architecture which provides a high-level, comprehensive approach for designing, planning, implementation and governance of enterprise information architecture [37]. TOGAF's taxonomy of architecture views defines the four categories of architectural views that should be considered in the development architecture. *Business Architecture Views* address the concerns of the users of the system. They describe the flows of business information between people and business processes. *Data Architecture Views* describes data entities and their interrelations addressing database designers' and administrators' concerns. *Application Architecture Views* provide blueprint for the system, its interactions to other systems. *Technical Architecture Views* describes the hardware, software and network infrastructure to support the application functionalities.

DoDAF

Department of Defense Architecture Framework (*DoDAF*) is an architecture framework that is targeted for United States Department of Defense that provides structure for a specific stakeholder through viewpoints organized by various views [9]. In DoDAF, architectural viewpoints are composed of data that has been organized to facilitate understanding. *All Viewpoint* describes the overarching aspects of architecture context that relate to all viewpoints. *Capability Viewpoint* articulates the capability requirements, delivery timing and deployed capability. *Data and Information Viewpoint* describes the data relationships in the architecture context of the architecture. *Operational Viewpoint* includes the operational scenarios, activities, and requirements that support capabilities. *Project Viewpoint* describes the relationships between operational and capability requirements. *Services Viewpoint* presents the design for solutions supporting operational and capability functions. *Standards Viewpoint* describes the operational, business, technical and industry policies, standards and constraints on system and service requirements. *Systems Viewpoint* describes the legacy support, the design for solutions articulating the systems, their interrelationships and compositions.

Chapter 3

Model-Driven Development

Historically, models have had a long tradition in software engineering and have been widely used in software projects. Software is a complex entity that is built upon both domain and technical knowledge. In order to be able to deal with only the relevant piece of the software at the desired level of abstraction, software researchers and developers create abstractions, namely *model*s of the software. Initially, models had been treated as only documentation. Model-Based Software Development (MBSD) aims to use models to develop software, but, it puts them into a completely separate place from the code. Recently, Model-Driven Software Development (MDSD) [35] paradigm entered to the stage which adopts models as the basic abstraction of software development process. According to MDSD, models are not only documentation but they can also directly participate into the code via automatic transformations.

In this chapter, we present the background on Model-Driven Development (MDD). Section 3.1 explains the concept of model. Section 3.2 gives basic information about metamodeling and software language engineering. Section 3.3 reports the value of model transformations in MDD and explains the two types of transformations: Model-to-text (M2T) and model-to-model (M2M) transformations.

3.1. Modeling

Modeling is a ubiquitous activity that we can observe in many areas of the real life. In general sense, a model is a set of statements that are used to describe the system under study. From this definition, it can be inferred that primary purpose of modeling is *describing* the subject entity.

In the context of software engineering, there exist several definitions of *model*. Here, we present some selected definitions that are collected by Muller et al. [29].

Definition 1

A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. [3]

Definition 2

Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones. [5]

Definition 3

A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made. [25]

We can summarize from these definitions a model is an abstraction of system that aim to **describe** a system from a **specific point of view, ignoring the unnecessary details**, providing a basis for **communication** and **analysis**. The highlighted properties make models valuable in the context of software engineering context.

Models are different in quality and nature. Mellor et al. [28] makes a distinction between three kinds of models depending on their level of precision.

Models as Sketches

Model as a sketch are used to communicate ideas and do not give much detail of the system. Sketches are informal diagrams used to communicate ideas, explore alternatives or design in a collaborative manner. They are usually focused on some aspect of the system and are not intended to show every detail of it.

Models as Blueprints

Model as a blueprint describe the system in sufficient detail. A blueprint must be enough to a programmer to code a system. In the case of forward engineering, the details of the blueprint should be enough for a programmer to code the system. In the case of reverse engineering, the diagrams show all the details of a system in order to understand it better or to provide views of the code in a graphical form.

Models as Executables

Executable models are models that have everything required to produce desired functionality of a domain. They are more precise than sketches and blueprints. They can be compiled by model compilers. For example, in case of UML, executable UML means that UML can also be used as a programming language. When used in this form, the whole system is specified in the UML, the diagrams are the code, and they are compiled directly into executable binaries.

In model-driven software development the concept of models can be considered as executable models as defined by the above characterization of Mellor et al. []. This is in contrast to model-based software development in which models are used as blueprints at the most.

3.2. Meta-Modeling

Model-driven software development is a paradigm in which the concept of *model* is the key abstraction. In contrast to model-based software development, in model-driven software development models are not mere documentation but become “code” that are executable and that can be used to generate even more refined models or code. MDSD aims to achieve this goal through defining *models* and *metamodels* as first class abstractions, and providing automated support using *model transformations* [14][28][34].

In the context of MDSD metamodeling plays a very important role. The language in which models are expressed is defined by metamodel. More precisely, a metamodel describes the possible structure of models in an abstract way. It defines the constructs of modeling language, their relationships, constraints and rules. A model is said to be an instance of a meta-model, or a model conforms to a meta-model. A meta-model itself is a model that conforms to a meta-meta-model, the language for defining meta-models. In model-driven development, models are usually organized in a four-layered architecture. The top (M3) level in this model is the so called meta-metamodel, and defines the basic concepts from which specific meta-models are created at the meta (M2) level. Normal user models are regarded as residing at the M1 level, whereas real world concepts reside at level M0

The four-layered architecture [30] can be explained better via the example in Figure 3.1. In the bottommost layer M0, the real concrete system to be described lies. In M1, the model layer, there is class diagram which is in fact a model of the real system. In metamodeling layer, the concepts to define a class diagram are presented. The language used for modeling the class diagram-UML lies in the M2 layer. The topmost layer is the meta-metamodeling layer which embodies the language for defining the metamodel, in this case UM which is meta-object facility(MOF). The topmost layer M3 is accepted to be recursively defined by itself.

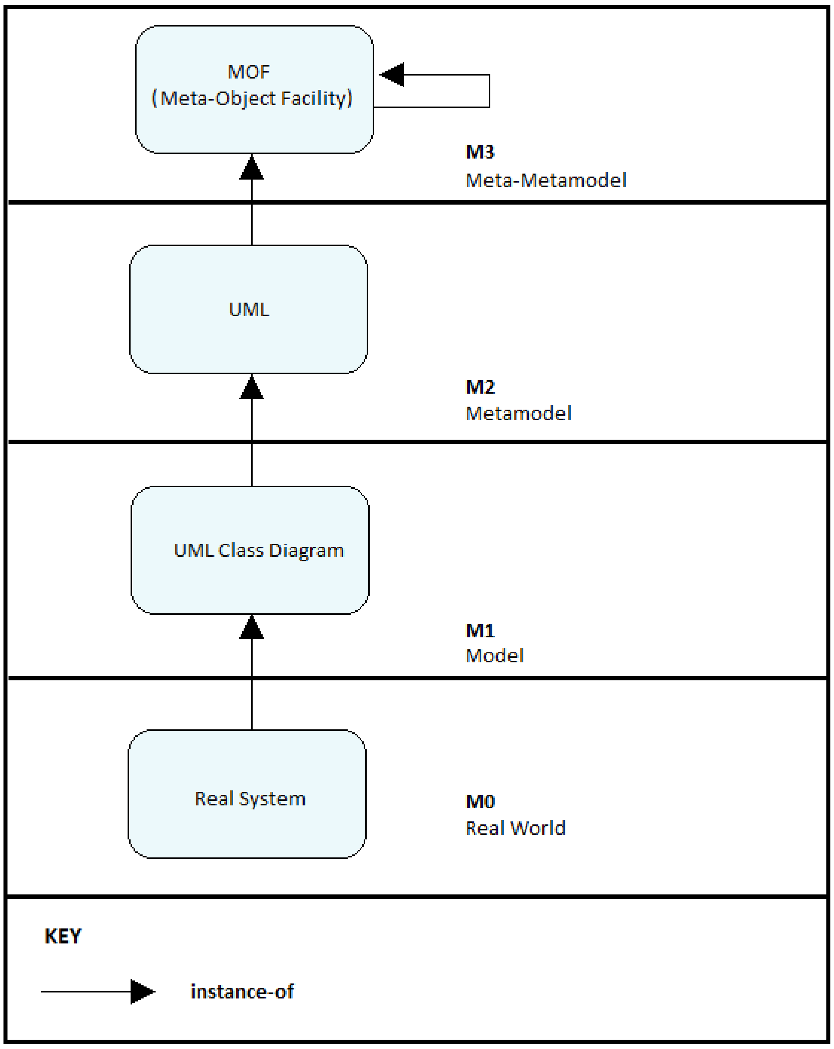


Figure 3.1. An example for the four-layer OMG architecture

Metamodels are important concepts in not only MDSD domain but also in software language engineering (SLE) [23] which is defined as the application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of languages. A proper definition of meta-models is important to enable valid and sound models. In both the software language engineering [23] and model-driven development domains [35], a meta-model should include the following elements:

Abstract Syntax: describes the vocabulary of concepts provided by the language and how they may be combined to create models. It consists of a definition of the concepts and the relationships that exist between concepts.

Concrete Syntax: defines the syntax, the notation that facilitates the presentation and construction of models or programs in the language. Typically two basic types of concrete syntax are used by languages: textual syntax and visual syntax. A textual syntax enables models to be described in a structured textual form. A visual syntax enables a model to be described in a diagrammatical form.

Well-formedness rules (Static Semantics): provides definitions of additional constraint rules on abstract syntax that are hard or impossible to express in standard syntactic formalisms of the abstract syntax.

Semantics: The description of the meaning of the concepts and relation in the abstract syntax. Semantics can be defined in natural language or using other more formal specification languages.

Figure 3.2 shows the elements that constitutes a metamodel and their relationships.

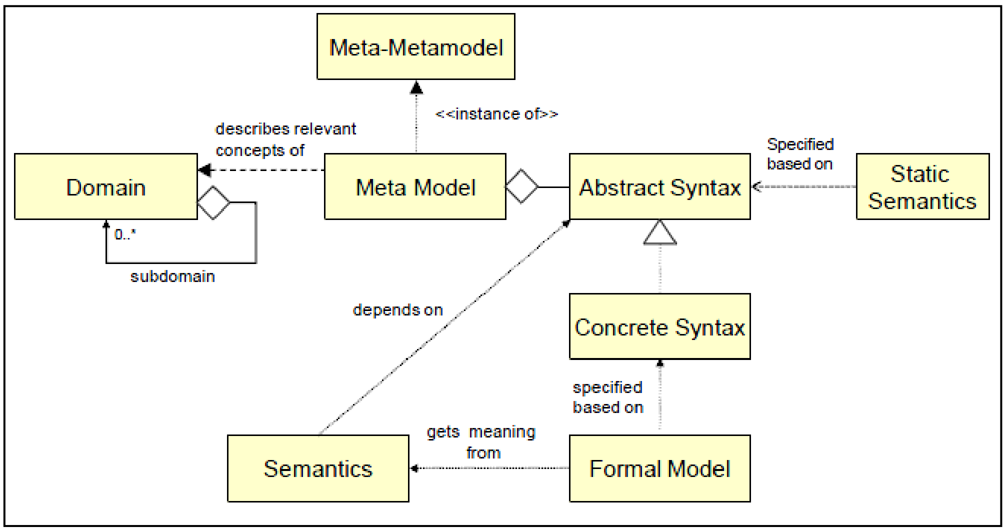


Figure 3.2. A conceptual model to describe metamodeling concepts.

3.3. Model Transformations

The notion of model transformation is central to model-driven engineering [28][35]. A model transformation takes a model conforming to a given metamodel as input and produces another model as output which also conforms to a given metamodel. Model transformations are useful for the following purposes:

- Generating lower-level models, eventually code, from higher level models
- Ensuring that a family of models is consistent, saving effort and reducing errors by automating the building and modification of models where possible
- Mapping and synchronizing among models at the same level of different levels of abstraction
- Reverse engineering of higher-level models from lower-level models or code.

Figure 3.3 explains basic model transformation pattern [35]. Source model is defined based on the source metamodel. There is also a given target metamodel. Transformation definition defines how a model conforming to source metamodel can be translated to an output model conforming to the target metamodel. The transformation definition is executed by a transformation engine. It reads the source model and outputs the target model. The transformation can be unidirectional or bidirectional based on the transformation definition.

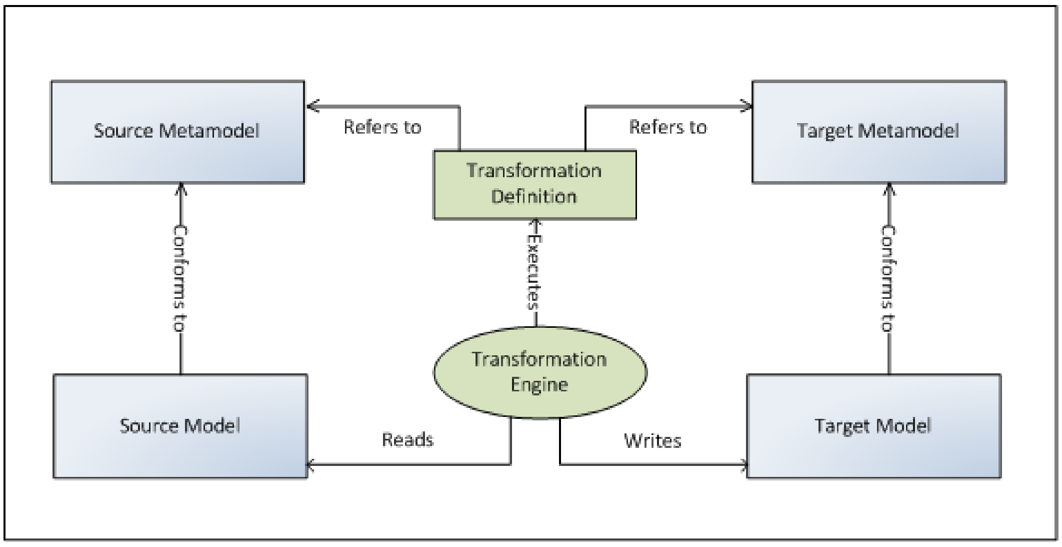


Figure 3.3. Model transformation pattern

Basically models transformations are categorized into two types: model-to-model and model-to-text transformations.

Model-to-Model (M2M) Transformation: In model-to-model transformation a model is transformed into another model (target model) which is instance of either the source metamodel or another metamodel. Both input and output are models which conforms to some metamodel. Transformation rules are defined to support M2M transformations and they are executed by transformation engine. Based on the definition of those rules, transformation can be unidirectional or bidirectional. The Eclipse M2M project provides a framework for model-to-model transformation languages including ATL[22], Operational QVT and Relational QVT [31]. Model-to-model transformation is required to ease generation of intermediate software models and keeping all models consistent.

Model-to-Text(M2T) Transformation: Model-to-text transformation which is also referred as model-to-code transformation is a special case of model-to-model transformation in which there is no target metamodel and the target output is a text. Model-to-text transformations are useful generating textual artifacts like code and documentation. It is standardized how to translate a model to various texts such as code, specifications, reports and documents in MOF Model to Text standard. Essentially, the standard needs to address how to transform a model into a linearized text representation. A template-based approach is defined in which the text to be generated from models is specified as a set of text templates that are parameterized with model elements. In the literature, there exists various tools that support model-to-text transformations that are developed based on MOF M2T standard such as Acceleo [1] and Xpand [38].

Chapter 4

Domain-Specific Languages for Software Architecture Viewpoints

Architectural views are usually developed based on architectural viewpoints which define the conventions for constructing, interpreting and analyzing views. Our analysis on the architectural viewpoints yields that so far most architectural viewpoints seem to have been primarily used either to support the communication among stakeholders, or at the best to provide a blueprint for the detailed design. They are not used as, executable architectural models. We identified that one important reason behind this is that the architectural viewpoints in the literature are not well and precisely defined. In order to address this problem, we propose that architectural viewpoints should be defined as domain-specific languages (DSL). In this chapter, we provide a software language engineering approach to define viewpoints as domain specific languages. We illustrate our approach with the viewpoints of the Views and Beyond framework using Crisis Management System (CMS) architecture as the case study. We also set up a framework to evaluate the quality of the viewpoint definitions. After defining all viewpoints as DSLs, we illustrate how the current viewpoint definitions are improved when they are defined as DSLs using the evaluation framework.

4.1. Viewpoints as Metamodels

In architecture modeling literature the notion of meta-model is not explicitly used. Nevertheless, the concepts related to architectural description are formalized and standardized in ISO/IEC 42010:2007, a fast-track adoption by ISO of IEEE-Std 1471-2000, Recommended Practice for Architecture Description of Software-Intensive Systems [20]. The standard holds that an architecture description consists of a set of views, each of which conforms to a viewpoint, but it has deliberately chosen not to define a particular viewpoint. Here the concept of view appears to be at the same level of to the concept of model in the model-driven development approach. The concept of viewpoint, representing the language for expressing views, appears to be on the level of meta-model.

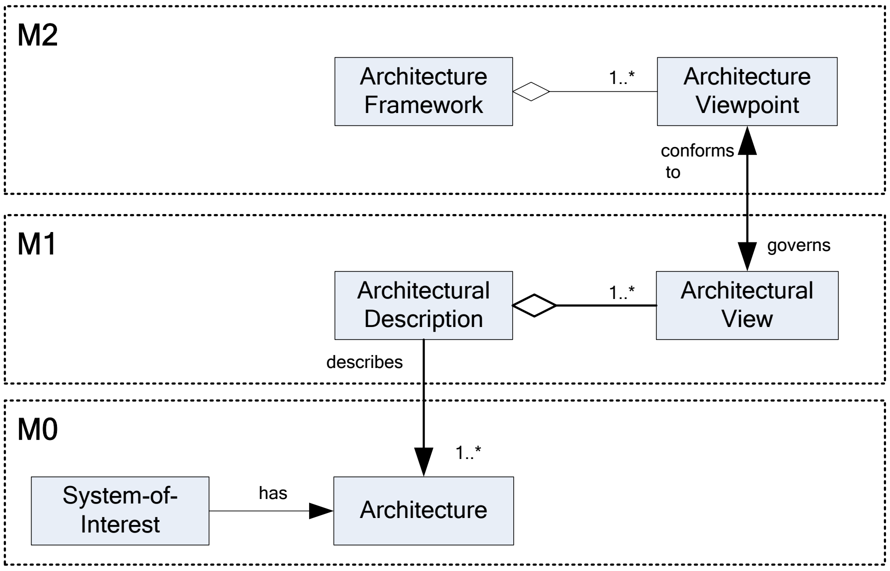


Figure 4.1. Architectural Description Concepts from a meta-modeling perspective

Although the ISO/IEC 42010 standard does not really use the terminology of model-driven development the concepts as described in the standard seem to align with the concepts in the meta-modeling framework. In Figure 4.1, we provide a partial view of the standard that has been organized around the meta-modeling framework. An Architecture Description is a concrete artifact that documents the Architecture of a System of Interest. The concepts System-of-Interest and Architecture reside at layer M0. System-of-Interest defines a system for which an Architecture is defined. Architecture is described using Architectural Description that resides at level M1. Architectural Description includes one or more Architectural Views that represent the system from particular stakeholder concern's perspective. Architectural views are described based on Architectural Viewpoint, the language for the corresponding view. Architectural Viewpoints are organized in Architectural Framework. The latter two reside at level M2. The standard does not provide a concept that we could consider at level M3, and as such we have omitted this in Figure 4.1.

The key premise of this thesis is viewing the architectural viewpoints as metamodel. We build our work on top of this. As we mentioned in Chapter 3, a metamodel in other words a domain-specific language consists of the following elements: abstract syntax, concrete syntax, static semantics and semantics. We keep semantics discussion out of the scope of the thesis and follow the process shown in Figure 4.2 to define DSLs for architectural viewpoints. The formal viewpoint definitions given in Chapter 4.3 are defined based on this process.

As we mentioned earlier, we selected Views and Beyond framework viewpoints to defined them as DSLs. For each DSL, we first present the abstract syntax that defines the language abstractions and their relationship. The abstract syntax for a viewpoint is defined after an analysis of the viewpoint description in the corresponding textbook [7].

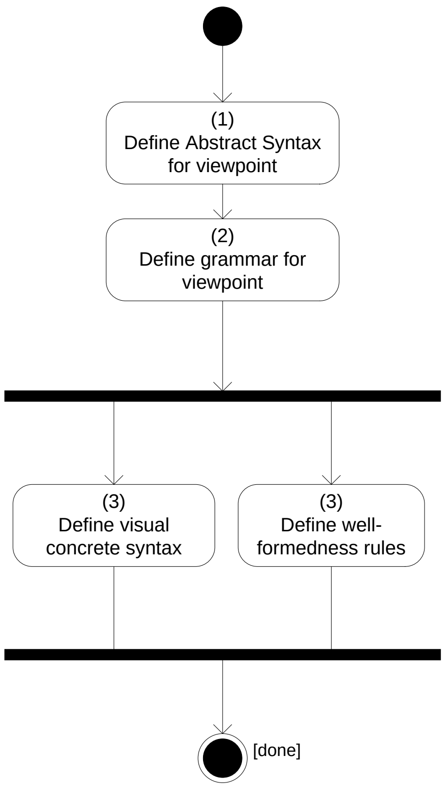

(1)
Define Abstract Syntax for viewpoint
(2)
Define grammar for viewpoint
(3)
Define visual concrete syntax
(3)
Define well-formedness rules
[done]

Figure 4.2. The process of defining DSLs for architectural viewpoints

Based on these descriptions and the defined meta-model we provide the grammar which defines syntactic rules of the language together with textual concrete syntax. The grammar is defined using Xtext a language development framework provided as an Eclipse plug-in [11]. The grammar of the language is defined in Xtext's [39] EBNF grammar language and the corresponding generator creates a parser, an AST-meta model (implemented in EMF) as well as a full-featured Eclipse Text Editor from that. After defining the grammar, we have our pure language at hand. We enrich our language defining visual concrete syntax and well-formedness rules. These two are not mandatory steps and none of them are prerequisite to each other. As shown in Figure 4.2, they can be done in parallel. The visual concrete syntax is defined using Graphical Modeling Framework (GMF) plug-in of Eclipse [16]. Constraints on viewpoint elements and relations are implemented as static semantics which is implemented writing validation codes in Java.

4.2. Case Study: Crisis Management System

In this section, we present the case study Crisis Management System (CMS) [21] for which we will define sample architectural views when illustrating our domain-specific languages for viewpoints in section 4.3.

A crisis management system is a software system that helps in:

- identifying, assessing, and handling a crisis situation
- by coordinating the communication between all parties involved in handling the crisis,
- by allocating and managing resources,
- and by providing access to relevant crisis-related information to authorized users.

The need for crisis management systems has grown significantly over time. In the context of CMS a crisis can be major event that affects various segments of society such as natural disasters, terrorist attacks or accidents. The role of crisis management system is to facilitate the process of resolving the crisis by coordinating the relevant parties.

A crisis management scenario is initiated by a *crisis report* from a *witness* at the *scene*. A *coordinator*, who is responsible for organizing all required *resources* and *tasks*, initiates the crisis management process. The coordinator has access to the camera surveillance system. If a crisis occurs in locations under surveillance, the crisis management system can request video feed that allows the coordinator to verify the witness information.

A *super observer* who is an expert depending on the kind of crisis, is assigned to the scene to observe the emergency situation and identify the tasks necessary to cope with the situation. The tasks are crisis missions defined by the observer. The coordinator is required to process the missions by allocating suitable resources to each task.

Depending on the type of crisis, human resources could include firemen, doctors, nurses, policemen, and technicians, and hardware resources could include transportation systems, computing resources, communication means, or other necessities like food or clothes. The human resources act as first-aid workers. Each first-aid worker is assigned a specific task which needs to be executed to recover from the abnormal situation. The workers are expected to report on the success or failure in carrying out the missions. The completion of all missions would allow the crisis to be concluded.

In summary, a crisis management system (CMS) should include the following functionalities:

- Initiating a crisis based on an external input from a witness,
- Processing a crisis by executing the missions defined by a super observer and then assigning internal and/or external resources,
- wrapping-up and archiving crisis,
- authenticating users,
- handling communication between coordinator/system and resources.

4.3. Domain Specific Languages for V&B Framework

In this section we will illustrate the modeling of viewpoints as domain specific languages to show how existing viewpoints can be even further formally specified to lift these to the level of executable models. We implement V&B framework [7] viewpoints as DSLs.

We will follow the process as defined in Figure 4.2. For each DSL, we first present the abstract syntax that defines the language abstractions and their relationship. Then, we provide the grammar which defines syntactic rules of the language together with textual concrete syntax. While presenting the language for a viewpoint, we provide example textual and visual view models of Crisis Management System that are defined using our DSLs.

4.3.1. Module Viewpoints

4.3.1.1. Decomposition Viewpoint

The Decomposition viewpoint [7] is used to show how system responsibilities are partitioned across modules and how these modules are decomposed into submodules. The decomposition view of the architecture depicts the overall structure of the architecture which is reasonably decomposed into modular implementation units. It is regarded as a fundamental view of the architecture since it serves as an input for other views (e.g. work allocation view) and helps to communicate and learn the structure of the software.

We have defined a DSL for decomposition viewpoint based on the textual specification given in [7]. The meta-model elements of it are provided below.

A model of the abstract syntax for the decomposition style is given in the left part of Figure 4.3. The root element is *DecompositionModel*. A valid decomposition model consists of elements. An element can either be a *Module* or *Subsystem*. *Module* denotes principal unit of implementation. *Subsystem* differs semantically from the module in the way that it can be developed, executed and deployed independent of other system parts. The decomposition relation between elements is established via the aggregation relation indicating that an element consists of other subelements. *Element* can have two types of properties: *Interface* and *Simple* property. The element’s interface is documented with interface property. An element’s interface can be declared as a reference to one of its children’s interface. Simple property is a generic property which allows specifying new properties in view document.

The grammar for decomposition style is given in the right part of Figure 4.3. An example decomposition view implemented using our DSL is shown in Figure 4.4. The textual concrete syntax is defined for elements. No explicit relation is modeled in order to express decomposition. Subelements are directly placed into the parent element.

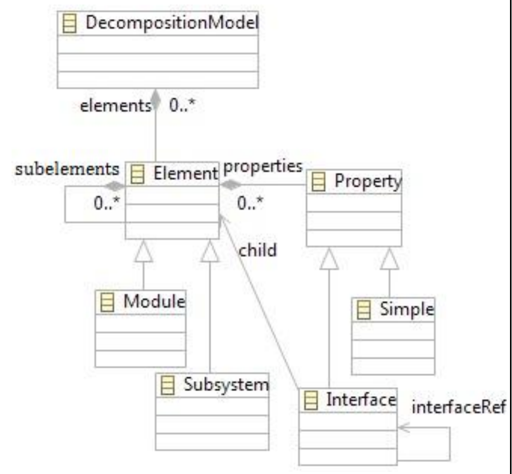
Abstract Syntax	Grammar
 DecompositionModel elements 0..* subelements Element properties Property 0..* 0..* child Module Simple Subsystem Interface interfaceRef	DecompositionModel: (elements += Element)*; Element: Module | Subsystem; Module: 'module' name=ID (('{' (properties+=Property)* (subelements+=Element)* '}') | ';'); Subsystem: 'subsystem' name=ID (('{' (properties+=Property)* (subelements+=Element)* '}') | ';'); Property: Simple | Interface; Interface: 'interface' (name=ID) ('=' child=[*Element*]'.'interfaceRef=[*Interface*])? ';'; Simple: 'property' feature=ID '=' value=STRING ';';

Figure 4.3. Abstract syntax and grammar for decomposition style

```
subsystem Crisis_Management_System{
    subsystem Crisis_Managemet_Subsystem{
        module Crisis_Reporting_Module;
        module Crisis_Handling_Module{
            module Task_Allocation;
            module Coordination;
        }
    }
    module Data_Management_Module;
    module Comm_Management_Module;
    module Offline_Reporting_Module;
}
```

Figure 4.4. Textual decomposition view model

Crisis Management System consists of one large subsystem, *Crisis Management Subsystem* and supplementary modules where *Comm Management* module establishes the communication infrastructure for the system, *Data Management* module utilizes DBMS operations in a modular way and *Offline Reporting* module enables taking various reports on the crisis events. *Crisis Management Subsystem* consists of *Crisis Reporting* module which enables initiating and maintaining crisis management process in a well-formed documented way and *Crisis Handling* module which enables taking task allocation and coordination actions to resolve crisis situation. Both textual and visual decomposition view models are easy-to-develop and understand. Visual view model for CMS decomposition viewpoint is given in Figure 4.5.

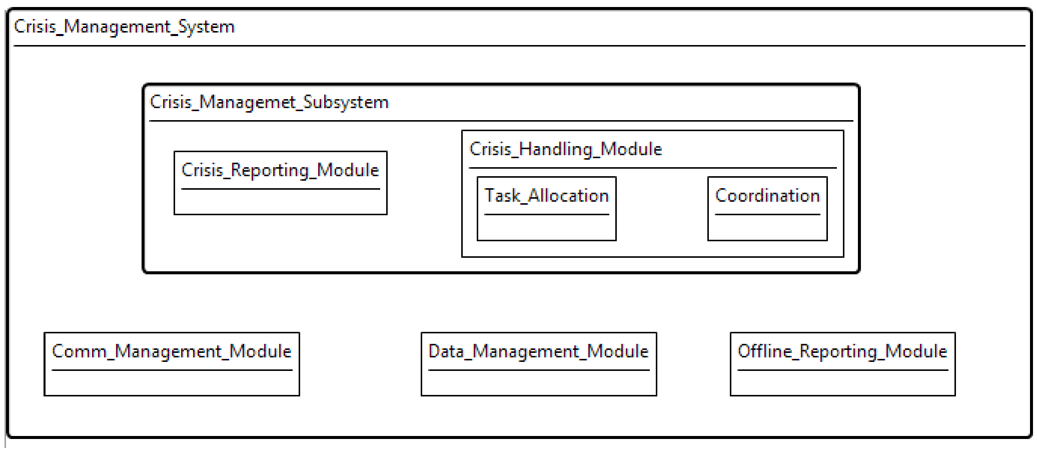


Figure 4.5. Visual decomposition view model

In addition to extracting the abstract syntax and the grammar we can also derive the well-formedness rules of views, the static semantics, from the viewpoint descriptions. In the decomposition style, two constraints have been defined: no loops are allowed in decomposition graph and a module can have only one parent. From the language perspective, those constraints are too high level to implement. We merged these constraints and shortly defined that no element can have the same name. Doing so we prevented both <A contains B, B contains A> case and <A contains B, C contains B> case. We implemented this constraint in Java as a validation rule that applies on the language model.

4.3.1.2. Uses Viewpoint

The uses viewpoint [7] results when the depends-on relation is specialized to uses. A module uses another module if its correctness depends on the correctness of the other. Uses viewpoint tells developers what other modules must exist for their portion of the system to work correctly. It enables incremental development and the deployment of useful subsets of full systems.

We have defined a DSL for uses viewpoint based on the textual specification given in [7]. The meta-model elements of it are provided below.

The root node of the grammar is *UsesModel*. It consists of *Elements* and *Uses Declaration* part. An *Element* of uses style is either a *Module* or a *Subsystem*. They are identified by their names. The relation is *Uses*. It has source and target attributes where both are references to *Element* instances. Figure 4.6 shows the abstract syntax for uses viewpoint and Figure 4.7 shows the grammar.

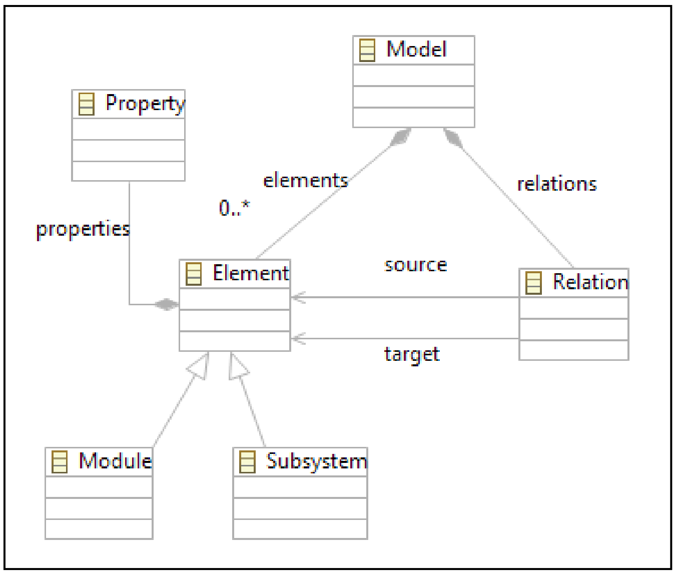
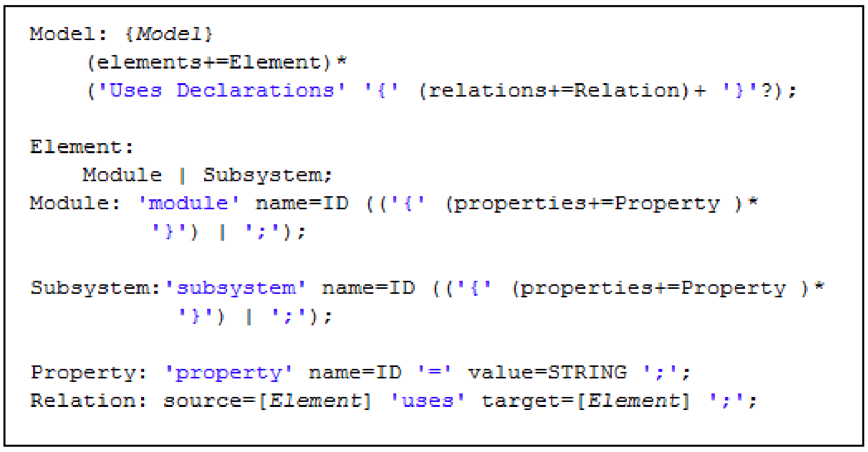


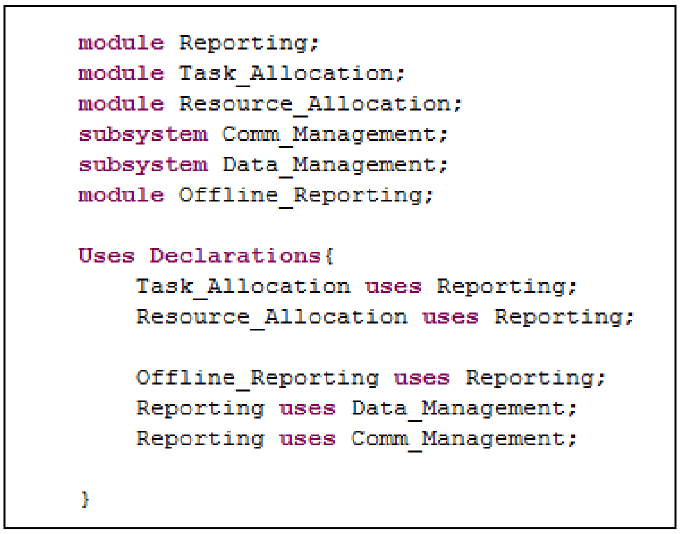
Figure 4.6. Abstract Syntax of Uses Viewpoint



```
Model: {Model}
     (elements+=Element)*
     ('Uses Declarations' '{' (relations+=Relation)+ '}'?);

Element:
     Module | Subsystem;
Module: 'module' name=ID (('{' (properties+=Property )*
          '}') | ';');

Subsystem:'subsystem' name=ID (('{' (properties+=Property )*
            '}') | ';');

Property: 'property' name=ID '=' value=STRING ';';
Relation: source=[Element] 'uses' target=[Element] ';';
```

Figure 4.7. Grammar of Uses Viewpoint

We have defined both textual and visual concrete syntax for uses viewpoint. In textual uses model, the subsystems and modules are listed by their names and uses declarations are specified in order to model the relation between those listed elements. In Figure 4.8, an example textual uses view is given for CMS. The modules and subsystems are listed first and then it is specified which modules uses the others. For example, *Task Allocation* and *Resource Allocation* modules uses *Reporting* modules which mean that crime reporting services must be correctly defined and implemented in order those two modules to be implemented.



```
module Reporting;
module Task_Allocation;
module Resource_Allocation;
subsystem Comm_Management;
subsystem Data_Management;
module Offline_Reporting;

Uses Declarations{
    Task_Allocation uses Reporting;
    Resource_Allocation uses Reporting;

    Offline_Reporting uses Reporting;
    Reporting uses Data_Management;
    Reporting uses Comm_Management;

}
```

Figure 4.8. Textual Uses View

The corresponding visual view model is also given in Figure 4.9.

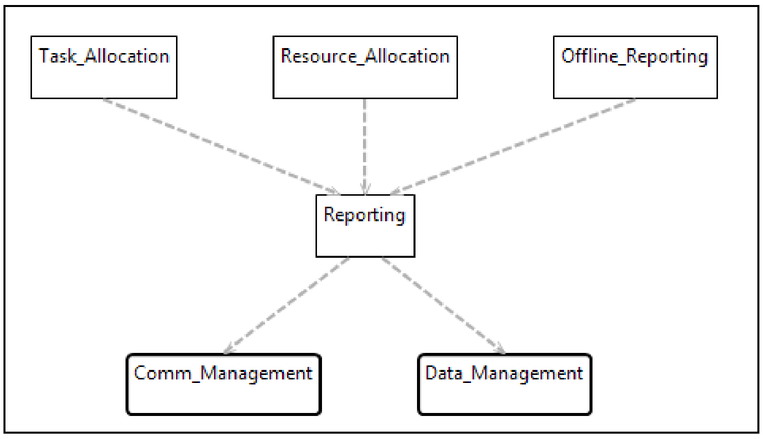


Figure 4.9. Visual uses view

Since there are no topological constraints for uses view, we didn’t implement any well-formedness rules for it.

4.3.1.3. Generalization Viewpoint

The generalization viewpoint [7] is useful for modeling is-a relation among modules. When an architect wants to support extension and evolution of architectures and individual elements, this viewpoint can be employed. Modules in this viewpoint are defined in such a way that they capture commonalities and variations. When modules have a generalization relationship, the parent module owns the commonalities, and the children modules own the variations

We have defined DSL for generalization viewpoint. The abstract syntax, grammar and textual and visual examples are given below.

The root node of the grammar is *GeneralizationModel*. It consists of *Element*s and *Generalization Declarations*. An element is either a *Module* or an *Interface*. They are identified by their names. Generalization declarations consist of *Relation*s. There are 3 types of relations: *InterfaceImpl*, *ClassInheritance, InterfaceInheritance*. If a module contains the implementation of an interface, it is denoted by *InterfaceImpl* relation. If a module inherits some behavior of other module, it is denoted by *ClassInheritance* relation. *InterfaceInheritance* denotes the definition of a new interface based on another previously defined interface. Figure 4.10 gives abstract syntax definition for generalization viewpoint.

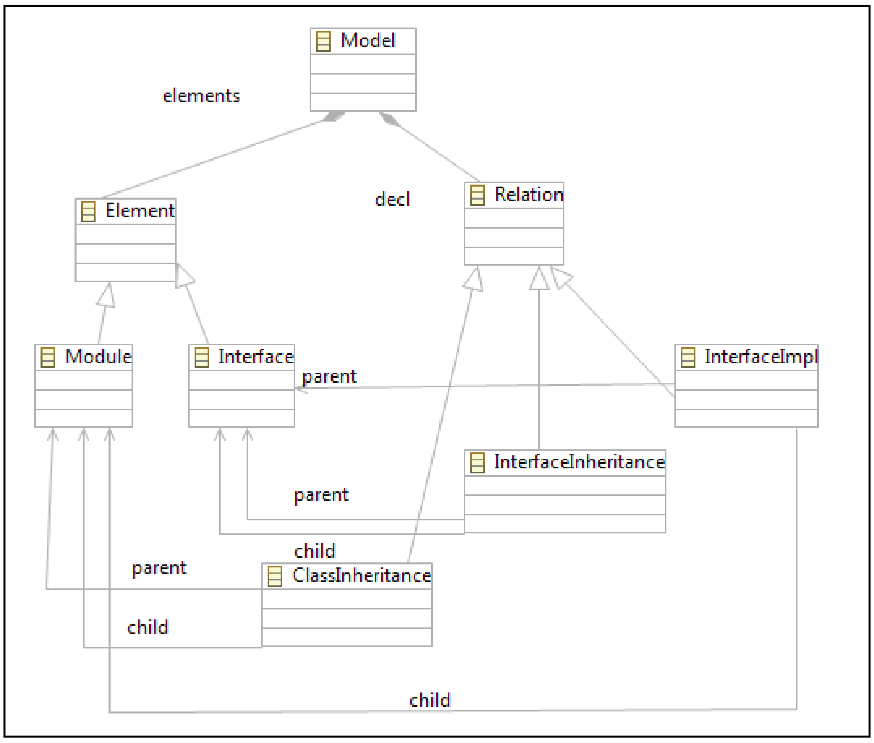


Figure 4.10. Abstract syntax for generalization viewpoint

In V&B generalization style, only module is defined as element type. If a module is an interface, it is denoted by “abstract” property of the module. Instead of differentiating modules and interfaces with a property, we defined Interface as a first-class abstraction in our grammar. V&B defines generalization relation as relation type and again motivates to differentiate different types of generalizations with properties. In our grammar, we explicitly define 3 types of relations InterfaceImpl, ClassInheritance, InterfaceInheritance. Figure 4.11 shows grammar for generalization viewpoint.

```
Model: {Model}
    (elements+=Element)*
    ('Generalization Declarations' '{'(decl+=Relation)* '}')?;

Element:
    Module | Interface;

Module:
    'module' name=ID ';';

Interface:
    'interface' name=ID ';';

Relation: InterfaceImpl | ClassInheritance | InterfaceInheritance;

InterfaceImpl: child=[Module] 'implements' 'interface' parent=[Interface] ';';
ClassInheritance: child=[Module] 'is-a' parent=[Module] ';';
InterfaceInheritance: 'interface' child=[Interface] 'is-a' 'interface' parent=[Interface] ';';
```

Figure 4.11. Grammar for generalization viewpoint

Below is an example generalization view from Crisis Management System. There is a generic module, namely interface, *Crisis Handler*, which includes the common properties that a specific type of crisis handler class must implement. After defining interfaces and modules the generalization relations are declared. For example, in our case, *Car Crash Handler* class implements *Crisis Handler* interface. The corresponding visual generalization view is also shown in Figure 4.13. The visual concrete syntax is very easy to understand. Modules and interfaces are differentiated from each other via their shapes and the Interface Implementation relation is denoted via an empty closed arrow.

```
interface Crisis_Handler;

module Car_Crash_Handler;
module Fire_Handler;
module Flood_Handler;

Generalization Declarations{
    Car_Crash_Handler implements interface Crisis_Handler;
    Fire_Handler implements interface Crisis_Handler;
    Flood_Handler implements interface Crisis_Handler;
}
```

Figure 4.12. Textual generalization view

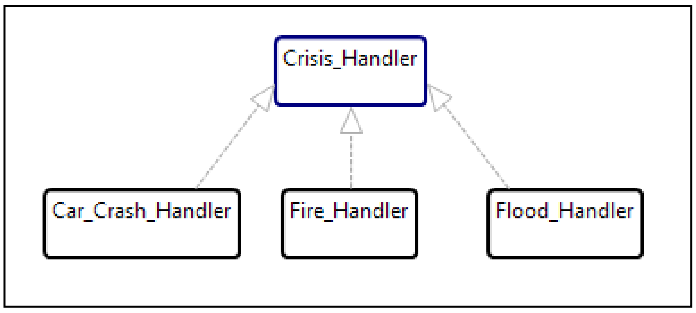


Figure 4.13. Visual generalization view

There is one topology constraint for generalization views. The relations cannot be cyclic or bi-directional. We have implemented this topology constraint as a well-formedness rule.

4.3.1.4. Layered Viewpoint

Another important module viewpoint defined in [7] is layered viewpoint. Layered viewpoint is similar to decomposition viewpoint since it reflects the division of software into units. The difference is that in layered viewpoint, layers, the first class elements of the style, are created to interact based on a strict ordering relation. If layer A is allowed to use layer B, layer A's implementation can use any public facilities of Layer B. However, layer B cannot use any facilities of layer A.

We have applied our language design process on layered viewpoint specification given in [7] to define a DSL for it.

The abstract syntax and grammar of layered viewpoint are given in Figure 4.14. The DSL for layered viewpoint consists of elements and relations. The element types are *Layer*s and *Segment*s. Semantically a layer is a group of software components that have similar module dependencies. The modules inside a layer are reusable in similar circumstances and they are likely to be ported to new applications together. Some layers consist of layer segments which are more cohesive subsets of layers. The fundamental relation of layered style is *allowed to use* relation. The ordering between layers is determined by allowed to use. There are two meaning of allowed to use relation. It can mean the source layer is allowed to use the target layer or source layer is allowed to use the target layer and all layers below it in the layering hierarchy. That's why, we defined two separate allowed to use relations: *allowed to use* and *allowed to use below*. There is no strict property that the layers have to exhibit. In case of any need to model some property of a layer, we define a generic property that can be defined for layers and layer segments.

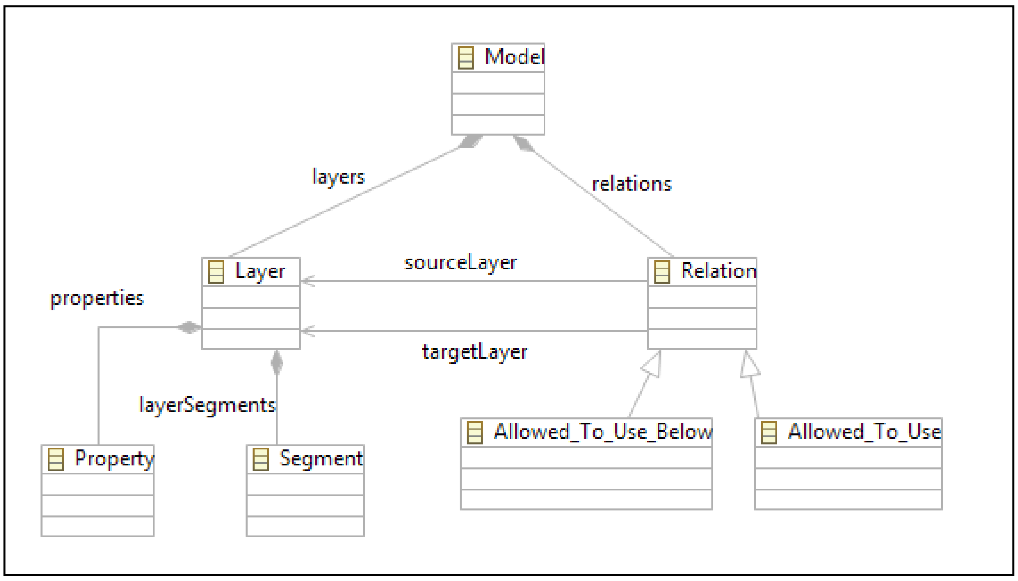
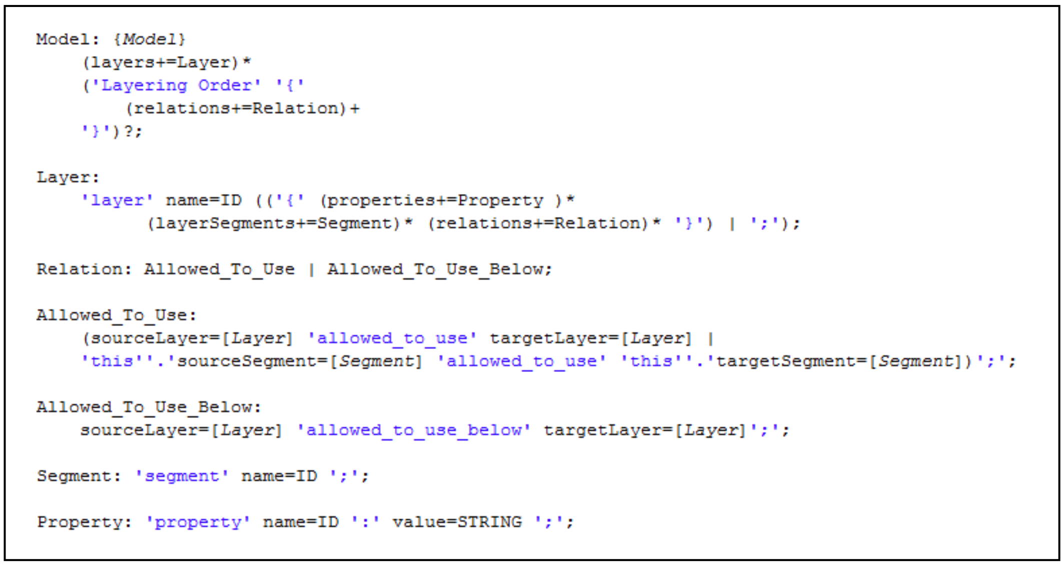


Figure 4.14. Abstract syntax for layered viewpoint

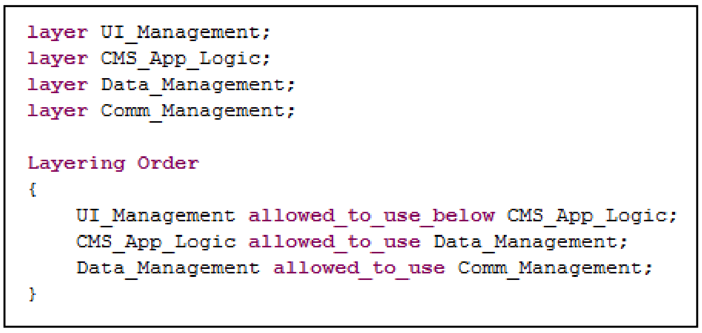
The grammar definition for layered viewpoint is given in Figure 4.15. The root node of the grammar is *LayeredModel*. It consists of layers and relations. A *Layer* is identified by its name and can contain properties and layer segments. *Layer segment*s are similar to layers except for they do not contain further layer segments. There are two types of relations: *allowed to use, allowed to use below*. *Allowed to use* relation is usually expressed in "[sourceLayer] allowed-to-use [targetLayer]" form. Allowed to use all below relation is expressed in "[sourceLayer] allowed to use [targetLayer] and below" form. A *property* is specified by its name which is an identifier and value which is a string following "property" keyword.



```
Model: {Model}
    (layers+=Layer)*
    ('Layering Order' '{'
        (relations+=Relation)+
    '}')?;

Layer:
    'layer' name=ID (('{' (properties+=Property )*
          (layerSegments+=Segment)* (relations+=Relation)* '}') | ';');

Relation: Allowed_To_Use | Allowed_To_Use_Below;

Allowed_To_Use:
    (sourceLayer=[Layer] 'allowed_to_use' targetLayer=[Layer] |
    'this''.'sourceSegment=[Segment] 'allowed_to_use' 'this''.'targetSegment=[Segment])';';

Allowed_To_Use_Below:
    sourceLayer=[Layer] 'allowed_to_use_below' targetLayer=[Layer]';';

Segment: 'segment' name=ID ';';

Property: 'property' name=ID ':' value=STRING ';';
```

Figure 4.15. Grammar for layered viewpoint

An example layered view of defined using the grammar is shown in Figure 4.16. On the top of the view model, layers are defined. Then layer ordering declarations starts where the relations are specified. The two types of relations both allowed to use and allowed to use below are used in the example view. *UI Management* module of CMS is allowed to use CMS application logic and all layers that it is allowed to use which are *Data Management* and *Comm Management* in our case.



```
layer UI_Management;
layer CMS_App_Logic;
layer Data_Management;
layer Comm_Management;

Layering Order
{
    UI_Management allowed_to_use_below CMS_App_Logic;
    CMS_App_Logic allowed_to_use Data_Management;
    Data_Management allowed_to_use Comm_Management;
}
```

Figure 4.16. Textual layered view

The visual concrete syntax for layered viewpoint consists of two element types: layers and layer segments. A layer is expressed by a rectangle where name of the layer is shown on the top of the rectangle. A layer segment is expressed by a rounded rectangle where its name is shown on top. The layer segments are directly placed inside layer figures. Visual concrete syntax is defined for two relations types: allowed to use and allowed to use below. Both are denoted by arrows from source layer to target layer. The allowed to use all below relation is expressed bolder than allowed to use in order to help to distinguish. Visual concrete syntax is not defined for properties in order to prevent crowd in diagram. An example layered view defined using visual concrete syntax is given in Figure 4.17.

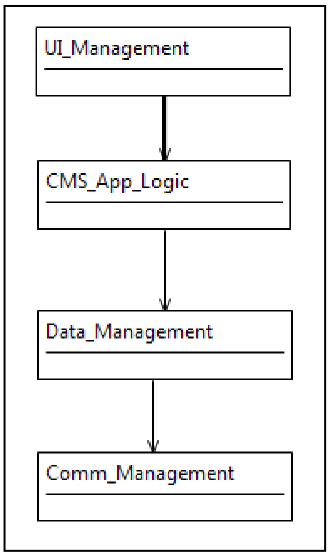


Figure 4.17. Visual layered view

The V&B layered style defines 3 constraints:

- Every piece of software is allocated to one layer.
- There are at least two layers
- Allowed to use relation cannot be circular.

The first constraint has two aspects: every module is allocated to a layer and every module is allocated to exactly one layer. The first is satisfied by the syntactic definition since we do not allow existence of modules outside the layers. For the second, we write a Java code that prevents giving same names to different modules.

The second constraint is partially satisfied by our grammar definition. We force the model to include at least one allowed to use relation. If layers are not allowed to use themselves this means the constraint is satisfied. We implement the constraint that prevents specifying allowed to use relation where target and source layers are the same.

The third constraint is more complex to implement. We implement it again writing a Java code. The layers that are allowed to use by some given source layer is found. Then, the layers allowed to use by previously identified layers are found and this process is repeated recursively. When recursion terminates, we have a tree structure at hand whose root node is initially given source layer. The tree is traversed for source layer and error is signaled if source layer is found in one of the tree's internal or leaf nodes.

4.3.1.5. Aspects Viewpoint

The aspects viewpoint [7] is a module viewpoint used to isolate in the architecture the modules responsible for crosscutting concerns. The viewpoint prescribes that the modules responsible for the crosscutting functionality should be placed in one or more aspect views. These modules are called aspects, based on the terminology introduced by aspect-oriented programming (AOP). The aspect views should contain information to bind each aspect module to the other modules that require the crosscutting functionality. The goal of designing and implementing crosscutting concerns in separate aspect modules is to improve modifiability of the modules that deal with the business domain functionality.

We have defined a DSL for aspects viewpoint and the metamodel is given below.

The root node of the grammar is *Model*. A model consists of *Element*s and *Crosscut Declarations*. An element can be either a *module* or an *aspect*. Both are identified by element names. Crosscut declarations are a set of statements that lists the aspects and shows which modules they crosscut in which way. A *crosscut* consists of an *Aspect* and *Detail*s. In details part, modules related to corresponding aspect are listed and it is given in which way the aspect crosscuts that module in a textual definition. Figure 4.18 and 4.19 shows the abstract sytax and the grammar for aspects view.

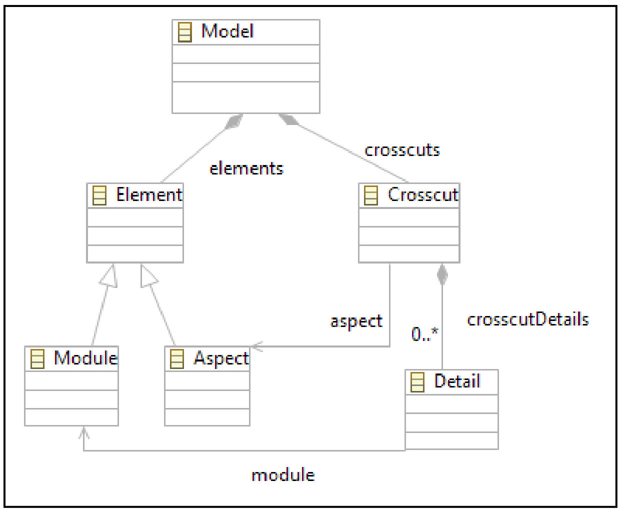
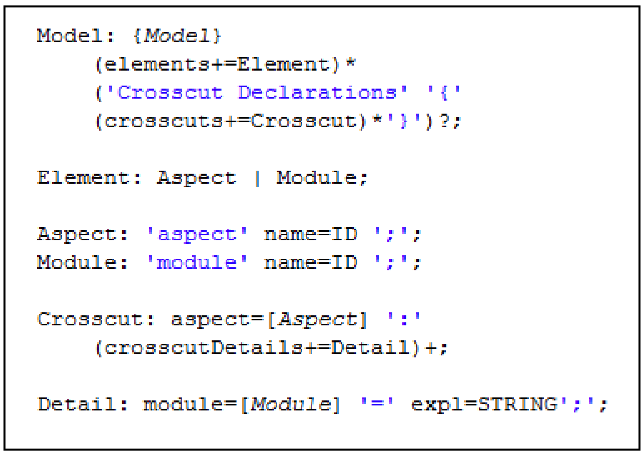


Figure 4.18. Abstract syntax for aspects viewpoint

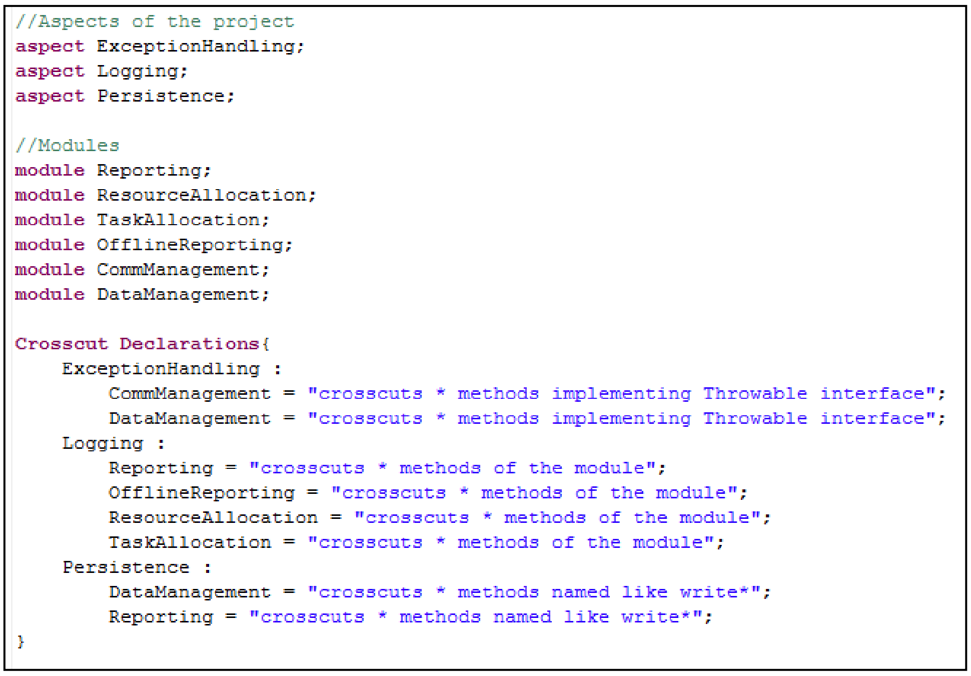


```
Model: {Model}
    (elements+=Element)*
    ('Crosscut Declarations' '{'
    (crosscuts+=Crosscut)*'}')?;

Element: Aspect | Module;

Aspect: 'aspect' name=ID ';';
Module: 'module' name=ID ';';

Crosscut: aspect=[Aspect] ':'
    (crosscutDetails+=Detail)+;

Detail: module=[Module] '=' expl=STRING';';
```

Figure 4.19. Grammar for aspects viewpoint

An example aspects view is given for CMS. *Logging*, ExceptionHandling, *Persistence* aspects crosscut various modules of the architecture. In the textual view, an explanation is provided for each crosscut declaration, however, we choose not to show those explanations in visual models to prevent crowd in the diagram. Figure 4.20 shows a textual aspects view of CMS architecture.

As topological constraint, we defined that there must definitely exist a crosscut declaration for each aspect and an aspect is related to at least one other module.



```
//Aspects of the project
aspect ExceptionHandling;
aspect Logging;
aspect Persistence;

//Modules
module Reporting;
module ResourceAllocation;
module TaskAllocation;
module OfflineReporting;
module CommManagement;
module DataManagement;

Crosscut Declarations{
    ExceptionHandling :
        CommManagement = "crosscuts * methods implementing Throwable interface";
        DataManagement = "crosscuts * methods implementing Throwable interface";
    Logging :
        Reporting = "crosscuts * methods of the module";
        OfflineReporting = "crosscuts * methods of the module";
        ResourceAllocation = "crosscuts * methods of the module";
        TaskAllocation = "crosscuts * methods of the module";
    Persistence :
        DataManagement = "crosscuts * methods named like write*";
        Reporting = "crosscuts * methods named like write*";
}
```

Figure 4.20. Textual aspects view

4.3.1.6. Data Model Viewpoint

Data modeling is a common activity in the software development process of information systems. The output of this activity is the data model, which describes the static information structure in terms of data entities and their relationships. The data model viewpoint [7] is prescribed in order to facilitate stakeholder communication during various stages of software development. Main uses of it are forming a model to communicate database optimization and normalization decisions, to reason about data access performance, to enable modifiability analysis or to ensure data integrity.

We implemented DSL for data model viewpoint and the metamodel of our DSL is explained below.

The first class abstraction of the language is *data entity*. A *data entity* describes any persistent object in the database. A data entity can have various *properties* such as data attributes and primary keys. The number and type of properties are open ended and left to view definer. There can be 3 types of relations: *associations*, *generalizations* and *aggregations* between data entities. The cardinality of associated entities (one-to-one, one-to-many etc.) must be defined when declaring association relation. The abstract syntax and grammar for data model view are given in Figure 4.21 and 4.22.

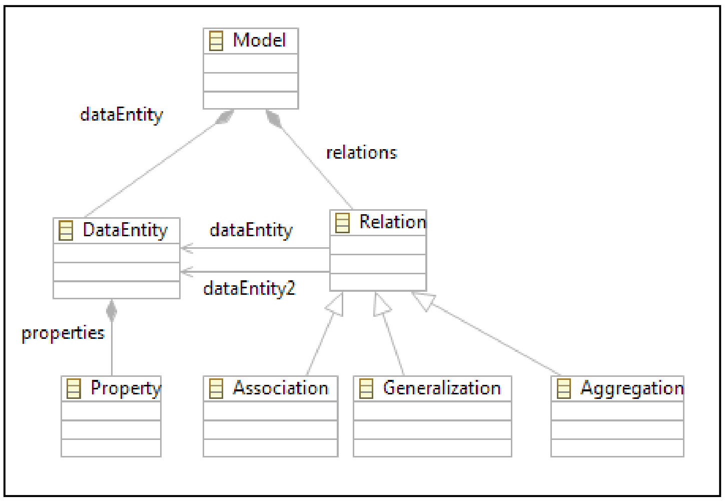
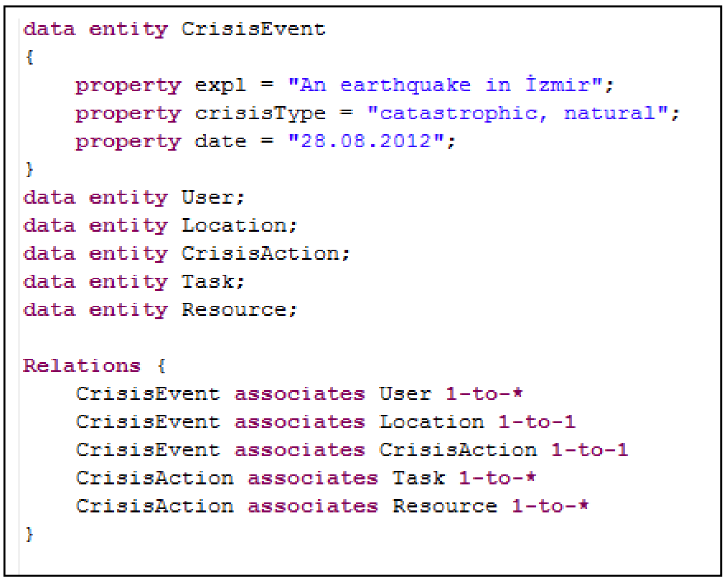


Figure 4.21. Abstract syntax for data model viewpoint

```
Model: {Model}
    (dataEntity+=DataEntity)*
    ('Relations' '{'
    (relations+=Relation)*'}')?;

DataEntity: 'data entity' name=ID
(('{' (properties+=Property )*'}') | ';');

Relation: Association | Generalization | Aggregation;

Association : dataEntity=[DataEntity] 'associates'
    dataEntity2 = [DataEntity] ('1-to-1'|'1-to-*'|'*-*');
Generalization : dataEntity=[DataEntity] 'is-a' dataEntity2 = [DataEntity];
Aggregation : dataEntity=[DataEntity] 'is-part-of' dataEntity2 = [DataEntity];

Property: 'property' name=ID '=' value=STRING ';';
```

Figure 4.22. Grammar for data model viewpoint

Figure 4.23 shows a textual data model view of CMS. The main entity here is *CrisisEvent*. A crisis event is described by an explanation, crisis type and date. Each crisis event can be related to one or more system users and each crisis event is related to at least one *Location*. There is a direct association to from *CrisisEvent* to *CrisisAction*. A crisis action consists of many tasks and many resources. The textual model is more detailed than visual view. We intentionally prefer not to model details like properties in visual data model in order to prevent crowd in the diagram.



```
data entity CrisisEvent
{
    property expl = "An earthquake in İzmir";
    property crisisType = "catastrophic, natural";
    property date = "28.08.2012";
}
data entity User;
data entity Location;
data entity CrisisAction;
data entity Task;
data entity Resource;

Relations {
    CrisisEvent associates User 1-to-*
    CrisisEvent associates Location 1-to-1
    CrisisEvent associates CrisisAction 1-to-1
    CrisisAction associates Task 1-to-*
    CrisisAction associates Resource 1-to-*
}
```

Figure 4.23. Textual data model view

There is no defined topological constraint in data model viewpoint. We identified that the is-a relations cannot be cyclic or bi-directional. The associates relation cannot be bidirectional.

4.3.2. Component and Connector Viewpoints

4.3.2.1. Pipe and Filter Viewpoint

The Pipe-and-Filter viewpoint [7] is a component-connector type style that shows the successive transformations on a stream of data. With their input ports, filter components take a stream of data, process it and direct to its output ports. Pipes are connectors between filter components. The pipe and filter style is usually used to model the data flow between run-time components of software, thus, the data dependencies between components can be identified and analyzed.

We have defined DSL for pipe-and-filter style of V&B approach.

Every component-connector style defines component and connector types. Component type that is used in pipe-and-filter style is *Filter*. A *filter* component gets data stream via its *input port* process the data stream and sends the resulting data over *output port*. Connector type defined for pipe-and-filter style is *Pipe*. *Pipe*s are responsible for connecting filters to each other, thus, enabling data flow between them. A *pipe* is a unidirectional connector between source and target filters. It has in and out roles. A *Pipe*'s in role is connected to output port of the source filter and out role is connected to input port of the target filter. The components in pipe and filter style can have *properties*. The abstract syntax for pipe-and-filter style derived based on the above explanations is shown in Figure 4.24.

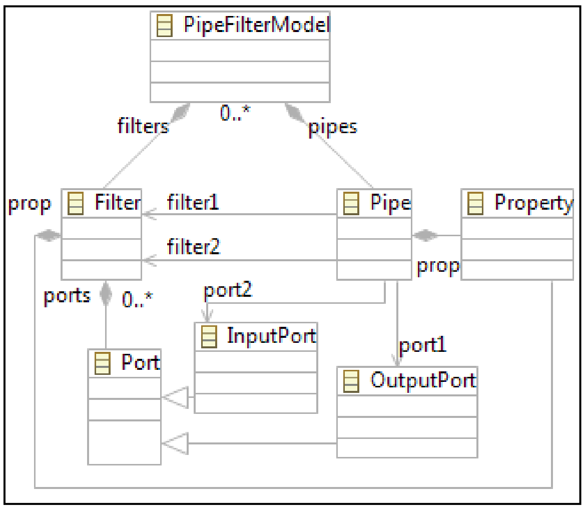
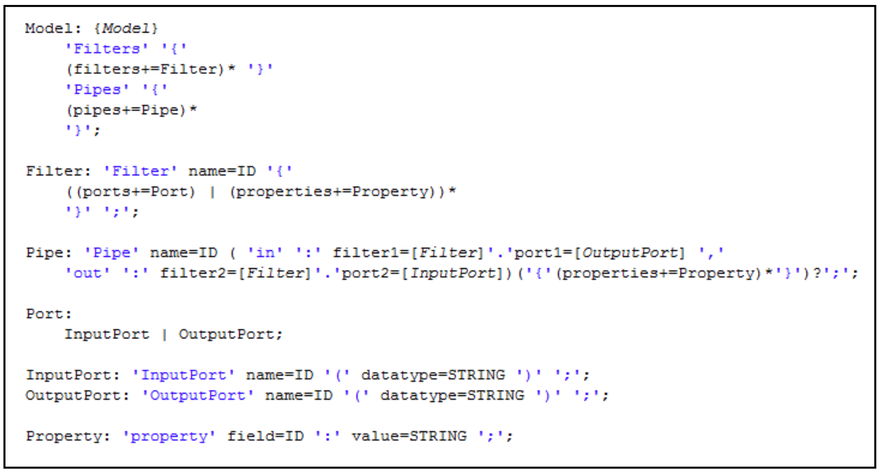


Figure 4.24. Abstract syntax for pipe-and-filter style

The grammar for pipe-and-filter style is given in Figure 4.25.



```
Model: {Model}
    'Filters' '{'
    (filters+=Filter)* '}'
    'Pipes' '{'
    (pipes+=Pipe)*
    '}';

Filter: 'Filter' name=ID '{'
    ((ports+=Port) | (properties+=Property))*
    '}' ';';

Pipe: 'Pipe' name=ID ( 'in' ':' filter1=[Filter]'.'port1=[OutputPort] ','
    'out' ':' filter2=[Filter]'.'port2=[InputPort])('{'(properties+=Property)*'}')?';';

Port:
    InputPort | OutputPort;

InputPort: 'InputPort' name=ID '(' datatype=STRING ')' ';';
OutputPort: 'OutputPort' name=ID '(' datatype=STRING ')' ';';

Property: 'property' field=ID ':' value=STRING ';';
```

Figure 4.26. Grammar for pipe-and-filter viewpoint

The root node of the grammar is *PipeFilterModel*. It consists of *Filter*s and *Pipe*s. *Filter* components are identified by their name. A filter must have at least one *Port*. A *Port* can be either an *InputPort* or an *Output Port*. Both either and output port must specify the data type that it can consume or produce. Pipes are connector components and they are identified by their names. A pipe has one in and one out attribute. The input of a *Pipe* is connected to *OutputPort* of a *Filter* and the output of a *Pipe* is connected to *InputPort* of a *Filter*. Both pipes and filters can have properties. There is no predefined property type; it is a generic model element. The property's name and value is defined at view level. An example pipe-and-filter view defined using the textual concrete syntax embedded in the grammar is seen in Figure 4.26. Note that first the filters are declared before pipes since pipe declarations require port information defined by filters.

The visual concrete syntax for pipe-and-filter style defines filters as elements explicitly showing their ports. The pipes are defined as directed associations that originate from a filter's output port to another filter's input port. The visual form of the example given in textual concrete syntax is shown in Figure 4.27.

In [] two important constraints are defined for pipe-and-filter style:

- Pipes connects filter output ports to filter input ports
- The connected filters must agree on the type of data being transferred.

```
Filters {
    Filter CrisisWatcher {
        OutputPort toHandler("stringBuffer");
        OutputPort toReporter("crisisObject");
    };
    Filter CrisisHandler{
        InputPort fromWatcher("stringBuffer");
        OutputPort toTaskAllocator("stringBuffer");
        OutputPort toResourceAllocator("stringBuffer");
    };
    Filter TaskAllocator{
        InputPort fromHandler("stringBuffer");
        OutputPort toReporter("taskList");
    };
    Filter ResourceAllocator{
        InputPort fromHandler("stringBuffer");
        OutputPort toReporter("resourceList");
    };
    Filter CrisisReporter{
        InputPort fromWatcher("crisisObject");
        InputPort fromTask("taskList");
        InputPort fromResource("resourceList");
    };
}
Pipes {
    Pipe w2h in : CrisisWatcher.toHandler, out: CrisisHandler.fromWatcher;
    Pipe w2r in : CrisisWatcher.toReporter, out: CrisisReporter.fromWatcher;
    Pipe h2t in : CrisisHandler.toTaskAllocator, out: TaskAllocator.fromHandler;
    Pipe h2r in : CrisisHandler.toResourceAllocator, out: ResourceAllocator.fromHandler;
    Pipe t2r in : TaskAllocator.toReporter, out: CrisisReporter.fromTask;
    Pipe rs2r in : ResourceAllocator.toReporter, out: CrisisReporter.fromResource;
```

Figure 4.26. Textual pipe-and-filter view

The first constraint is satisfied with our abstract syntax definition. A pipe is associated with an input and an output ports. For the second constraint we write a Java validation code that checks the datatype attributes of ports and signals error in case of any incompatibility.

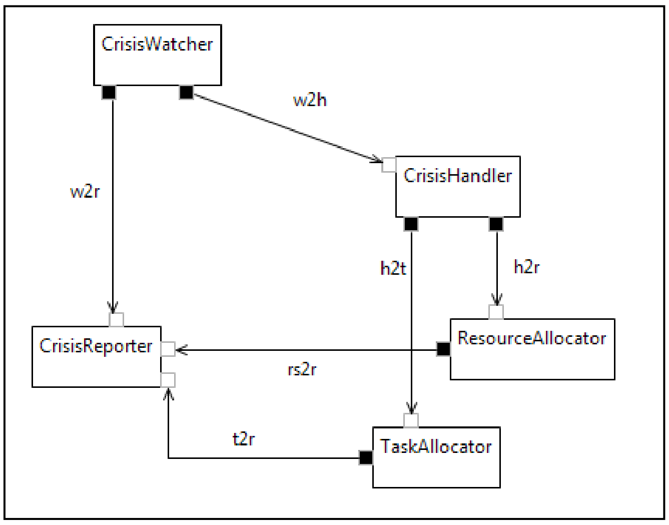


Figure 4.27. Visual pipe-and-filter view

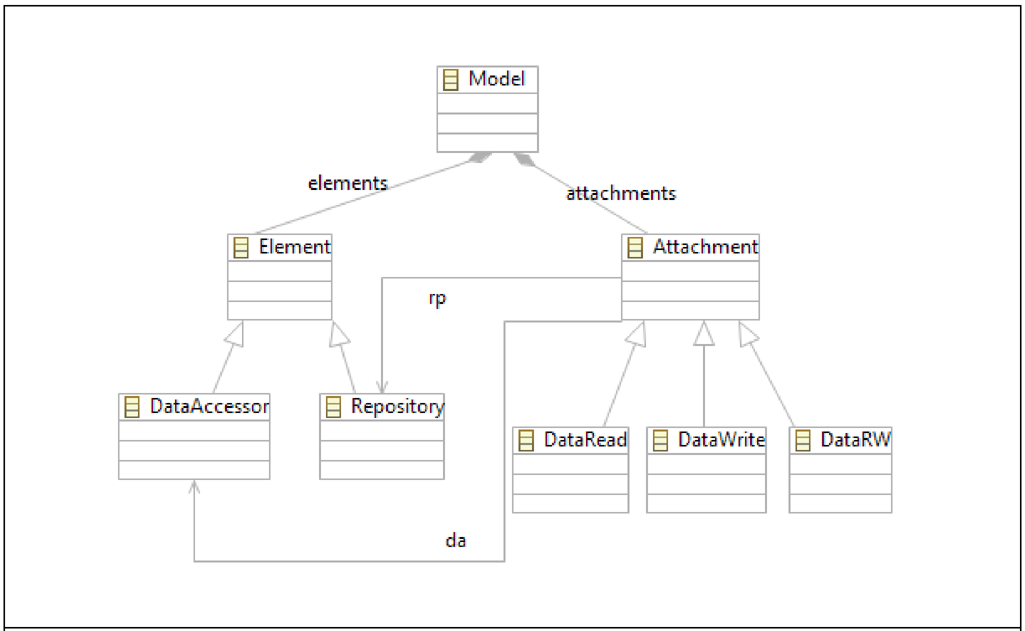
In addition to these constraints we define one more rule that prevents creating pipes when source and target filter is the same. We check this constraint by a simple Java code which signals error when a pipe is created with the same source and target filter.

4.3.2.2. Shared Data Viewpoint

Shared data viewpoint is another component-and-connector viewpoint which enables modeling the shared data repositories and the components that accesses those repositories together with their interaction. The viewpoint is useful whenever various data items have multiple accessors and persistence. Use of this viewpoint guides decoupling the producer of the data from the consumers of the data; hence this viewpoint supports modifiability, as the producers do not have direct knowledge of the consumers. Shared data view of a system supports analyses associated with performance, security, privacy, availability, scalability [7].

We have defined DSL for shared data viewpoint and the metamodel for it is given and explained below.

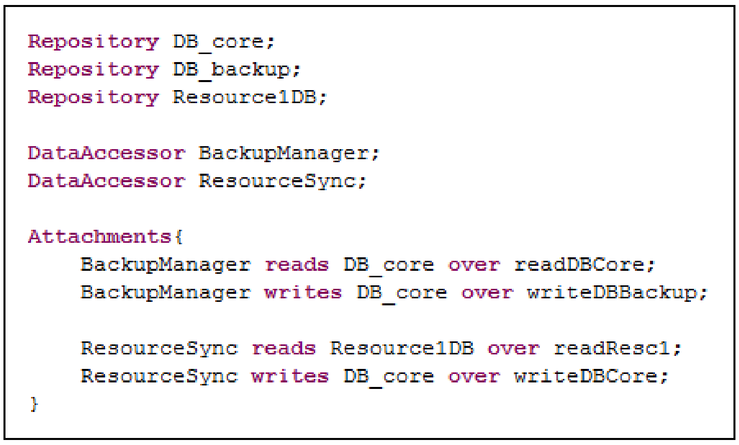
A shared data model consists of *Element*s and *Attachment*s. An element can either be *data accesor* or a *repository*. *Data accessor*s are attached to repositories by attachment relations which are named based on the purpose of the data access. An attachment can be *data read*, *data write* or *data read/write*. The grammar and abstract syntax model of shared data viewpoint are given in Figure 4.28.



```
Model: {Model}
    (elements+=Element)* 'Attachments' '{' (attachments+=Attachment)* '}';

Element: DataAccessor | Repository;

DataAccessor: 'DataAccessor' name=ID ';';
Repository: 'Repository' name=ID ';';

Attachment: DataRead | DataWrite | DataRW;

DataRead:
    da=[DataAccessor] 'reads' rp=[Repository] 'over' dataRead=ID ';';
DataWrite:
    da=[DataAccessor] 'writes' rp=[Repository] 'over' dataWrite=ID';';
DataRW:
    da=[DataAccessor] 'reads/writes' rp=[Repository] 'over' dataReadWrite=ID';';
```

Figure 4.28. Abstract syntax and grammar for shared data viewpoint

An example shared data view for CMS is given in Figure 4.29. There are three data repositories in CMS system model. The application logic runs on CMS *DB core*. There is a back up for the main CMS DB and *BackupManager* process reads CMS *DB_core* and writes on *DB_backup*. For crisis resolution, CMS system is integrated with various organizations' databases to know the current situation of the resources. Here in the model, we modeled hospital repository as *Resource1DB*. *ResourceSync* scripts run on a periodical manner to keep CMS database up-to-date about the current situation of the hospital. It reads from outsource database and writes into CMS main database.



```
Repository DB_core;
Repository DB_backup;
Repository Resource1DB;

DataAccessor BackupManager;
DataAccessor ResourceSync;

Attachments{
    BackupManager reads DB_core over readDBCore;
    BackupManager writes DB_core over writeDBBackup;

    ResourceSync reads Resource1DB over readResc1;
    ResourceSync writes DB_core over writeDBCore;
}
```

Figure 4.29. Textual shared data view

The topological constraints for shared data views are as follows:

- There must be at least one data accessor for each repository
- Each data accessor accesses at least one repository
- A data accessor must have at least one read and one write data access relation.

We have defined these topological constraints as well-formedness rules.

4.3.2.3. Publish-Subscribe Viewpoint

Publish-subscribe viewpoint [7] is defined to design event-based programs in a loosely coupled way by isolating event producers and event consumers from each other. In the publish-subscribe viewpoint, components interact via announced events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus the main form of connector in this style is a kind of event bus. Components place events on the bus by announcing them; the connector then delivers those events to the components that have registered an interest in those events. The computational model for the publish-subscribe style is best thought of as a system of independent processes or objects, which react to events generated by their environment, and which in turn cause reactions in other components as a side effect of their event announcements.

We have defined DSL for publish-subscribe viewpoint and the metamodel for it is explained below.

A publish-subscribe view consists of *Components*, at least one *Event Bus* as connector and *Attachment* relations. A component is connected to an event bus via an attachment. Attachment can be either a *Publish* or *Subscribe*. Both components and buses have input/output ports over which they communicate. The grammar and abstract syntax for publish-subscribe viewpoint is given in Figure 4.30, 4.31.

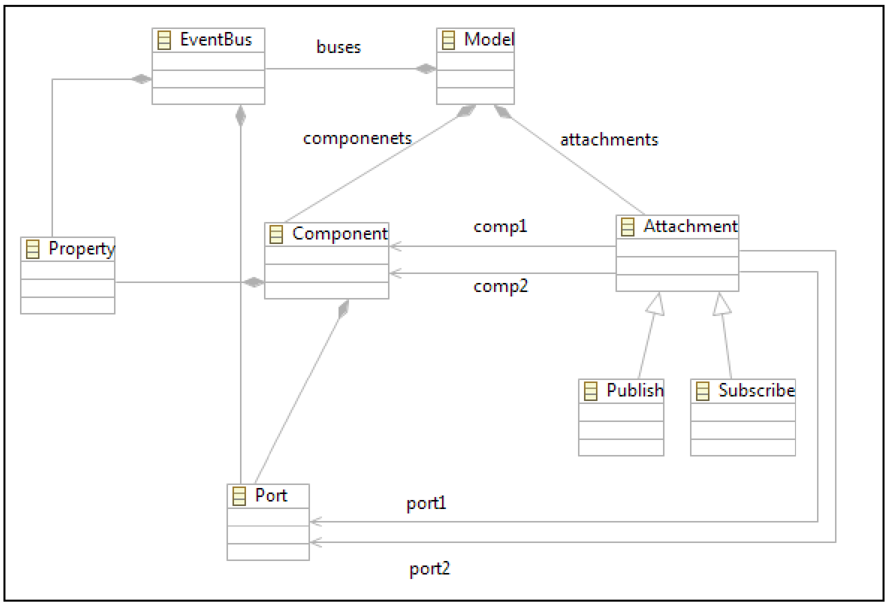
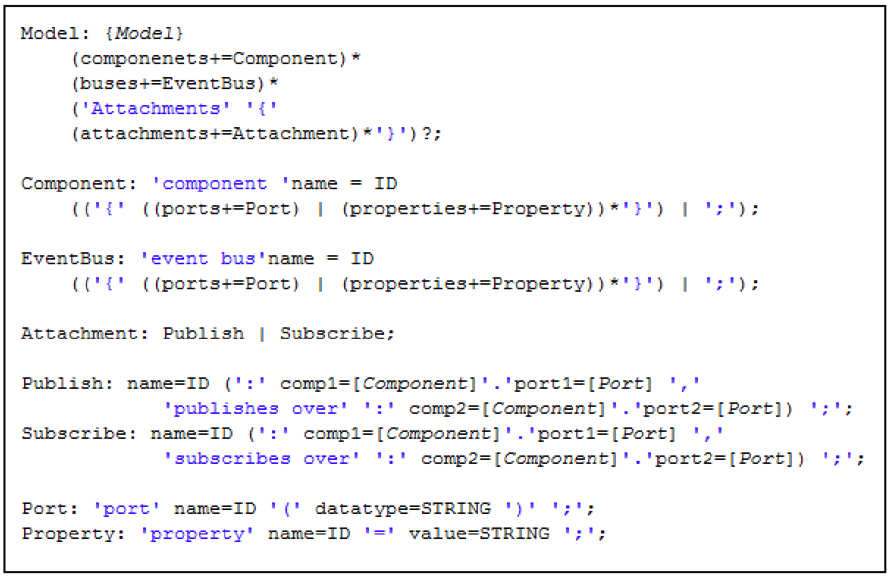


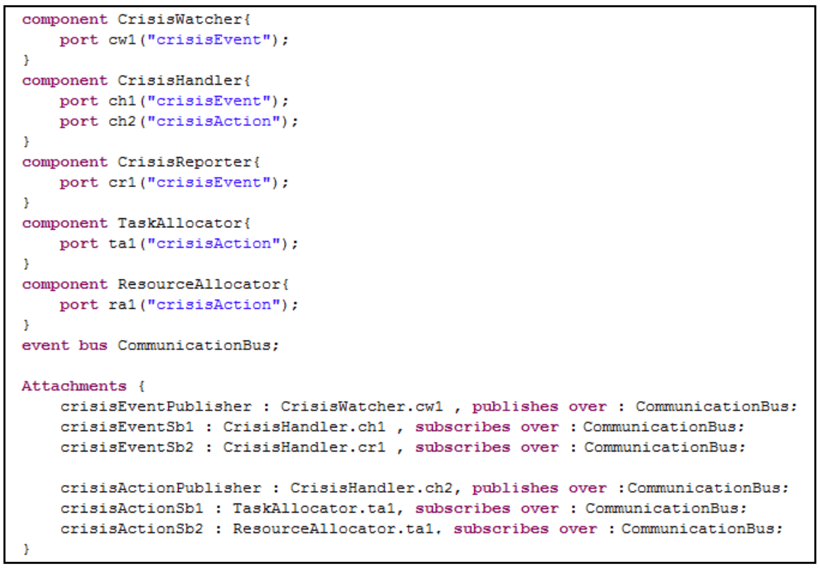
Figure 4.30. Abstract syntax for publish subscribe viewpoint



```
Model: {Model}
    (componenets+=Component)*
    (buses+=EventBus)*
    ('Attachments' '{'
    (attachments+=Attachment)*'}')?;

Component: 'component 'name = ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

EventBus: 'event bus'name = ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

Attachment: Publish | Subscribe;

Publish: name=ID (':' comp1=[Component]'.'port1=[Port] ','
           'publishes over' ':' comp2=[Component]'.'port2=[Port]) ';';
Subscribe: name=ID (':' comp1=[Component]'.'port1=[Port] ','
           'subscribes over' ':' comp2=[Component]'.'port2=[Port]) ';';

Port: 'port' name=ID '(' datatype=STRING ')' ';';
Property: 'property' name=ID '=' value=STRING ';';
```

Figure 4.31. Grammar for publish subscribe viewpoint

Figure 4.32 shows an example publish-subscribe view of CMS that is defined using our DSL. The common bus that enabled communication among processes is *Communication Bus*. *Crisis Watcher* publishes a crisis event to the bus. The bus is responsible for distributing the event to *Crisis Handler* and *Crisis Reporter* components which are subscribed to any crisis event. In similar way, *Crisis Handler* publishes crisis resolution action and the subscriber components *Task Allocator* and *Resource Allocator* are notified.



```
component CrisisWatcher{
    port cw1("crisisEvent");
}
component CrisisHandler{
    port ch1("crisisEvent");
    port ch2("crisisAction");
}
component CrisisReporter{
    port cr1("crisisEvent");
}
component TaskAllocator{
    port ta1("crisisAction");
}
component ResourceAllocator{
    port ra1("crisisAction");
}
event bus CommunicationBus;

Attachments {
    crisisEventPublisher : CrisisWatcher.cw1 , publishes over : CommunicationBus:
    crisisEventSb1 : CrisisHandler.ch1 , subscribes over : CommunicationBus:
    crisisEventSb2 : CrisisHandler.cr1 , subscribes over : CommunicationBus:

    crisisActionPublisher : CrisisHandler.ch2, publishes over :CommunicationBus:
    crisisActionSb1 : TaskAllocator.ta1, subscribes over : CommunicationBus:
    crisisActionSb2 : ResourceAllocator.ta1, subscribes over : CommunicationBus:
}
```

Figure 4.32. Textual publish-subscribe view

As static semantics of the language we identified that there must be at least one bus in the view. Components are not attached to each other directly, they are attached to the buses via publish and subscribe ports. The connected ports' data types must be compatible.

4.3.2.4. Client-Server Viewpoint

The client-server viewpoint presents a system view that separates client applications from the services they use [7]. It supports system understanding and reuse by factoring out common services. Because servers can be accessed by any number of clients, it is relatively easy to add new clients to a system. Similarly, servers may be replicated to support scalability or availability.

We have defined DSL for client-server viewpoint and the metamodel for it is explained below.

A client-server model simply consists of *Client* components, *Server* components and *Attachment*s those connect clients to servers. Clients have input ports, servers have output ports. Each client must be connected to at least one server. A client can be attached to multiple servers via distinct ports. There must not be more than one attachment relation between a specific client and a specific server. The grammar and abstract syntax for the language are given in Figure 4.33, 4.34.

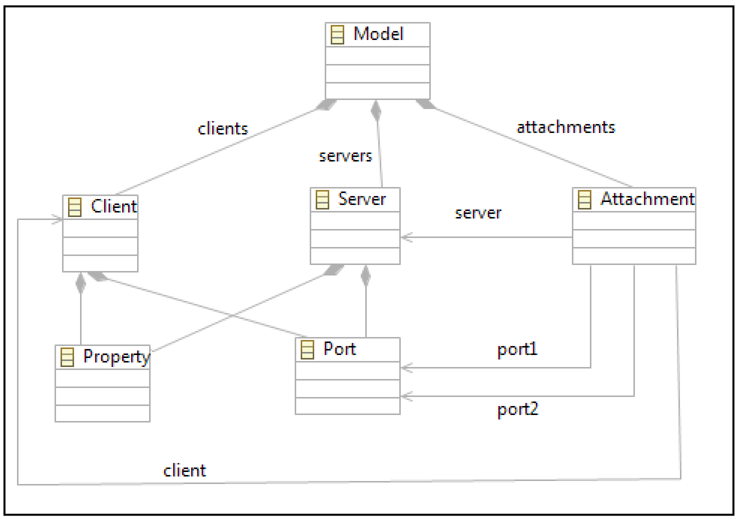
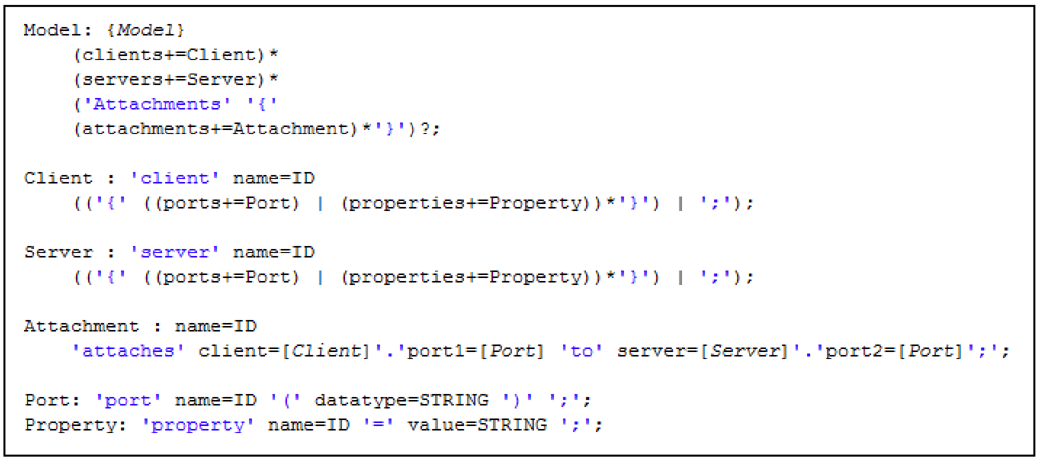


Figure 4.33. Abstract syntax for client-server viewpoint



```
Model: {Model}
    (clients+=Client)*
    (servers+=Server)*
    ('Attachments' '{'
    (attachments+=Attachment)*'}')?;

Client : 'client' name=ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

Server : 'server' name=ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

Attachment : name=ID
    'attaches' client=[Client]'.'port1=[Port] 'to' server=[Server]'.'port2=[Port]';';

Port: 'port' name=ID '(' datatype=STRING ')' ';';
Property: 'property' name=ID '=' value=STRING ';';
```

Figure 4.34. Grammar for client-server viewpoint

Figure 4.35 shows a simple client-server view of CMS architecture. The server component is *CMS Application Server*. Two types of clients can connect to the server in order to initiate a crisis report, *Mobile Client* and *Web Client*. *Crisis Handler* component is also a client to application server. The visual notation is simple, easy to model and understand.

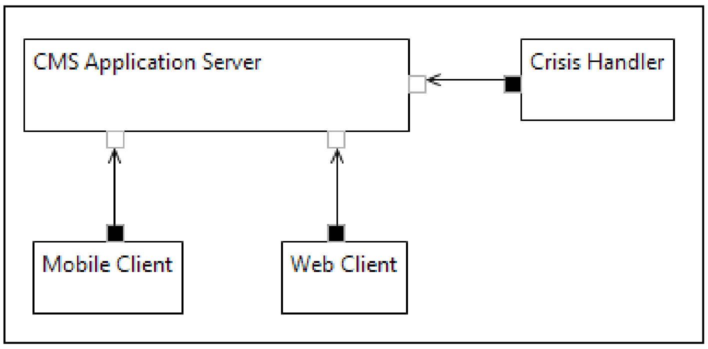


Figure 4.35. Visual client-server view

As static semantics of the language we identified that there must be at least one server and one client in the view. Components are not attached to each other directly, they are attached via ports. The connected ports' data types must be compatible.

4.3.2.5. Peer-to-Peer Viewpoint

In the peer-to-peer viewpoint [7], components directly interact as peers by exchanging services. Peer-to-peer communication is a kind of request/reply interaction without the asymmetry found in the client-server viewpoint. That is, any component can, in principle, interact with any other component by requesting its services. Each peer component provides and consumes similar services, and sometimes all peers are instances of the same component type. Connectors in peer-to-peer systems may involve complex bidirectional protocols of interaction, reflecting the two-way communication that may exist between two or more peer-to-peer components.

We have defined DSL for peer-to-peer viewpoint and the metamodel is given below.

The abstract syntax and grammar for peer-to-peer viewpoint is given in Figure 4.36. A peer-to-peer view model consists of elements and attachment relations. An element is either a *Peer* or a *Connector*. The peers can directly be attached to the connector via their *port*s. Requested services and provided services must be defined in the port declarations. There must be at least one connector in the view model.

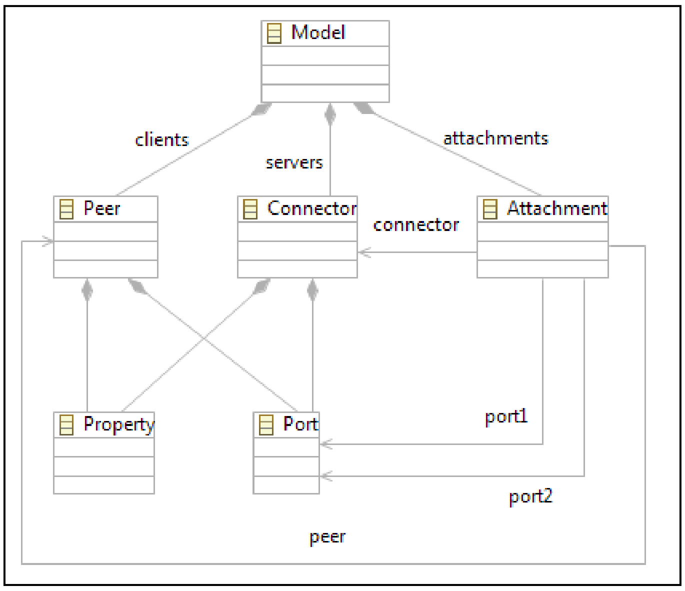
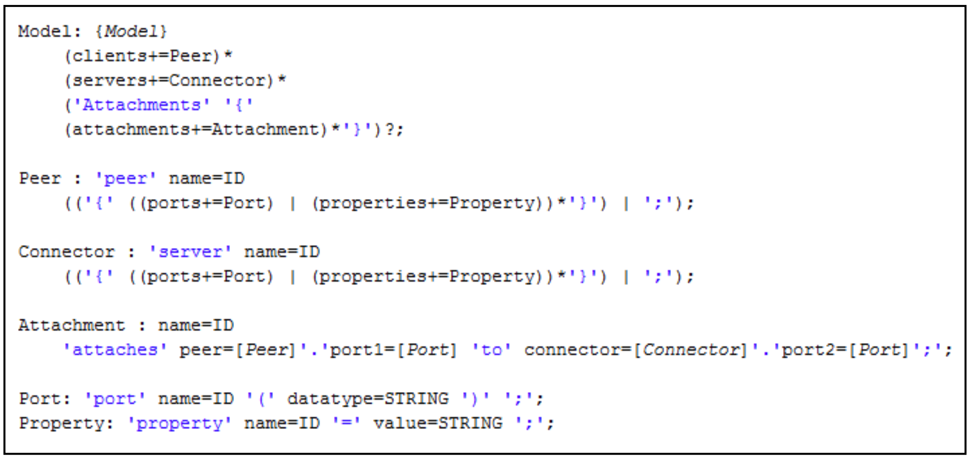


Figure 4.36. Abstract syntax for peer-to-peer viewpoint



```
Model: {Model}
    (clients+=Peer)*
    (servers+=Connector)*
    ('Attachments' '{'
    (attachments+=Attachment)*'}')?;

Peer : 'peer' name=ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

Connector : 'server' name=ID
    (('{' ((ports+=Port) | (properties+=Property))*'}') | ';');

Attachment : name=ID
    'attaches' peer=[Peer]'.'port1=[Port] 'to' connector=[Connector]'.'port2=[Port]';';

Port: 'port' name=ID '(' datatype=STRING ')' ';';
Property: 'property' name=ID '=' value=STRING ';';
```

Figure 4.37. Grammar for peer-to-peer viewpoint

A visual peer-to-peer view of CMS architecture is shown in Figure 4.38. The components are *Crisis Resolver*, *Task Allocator* and *Resource Allocator*. They communicate view *Peer Connector*. *Crisis Resolver* requests task services and resource services from the network which are produced by *Task Allocator* and *Resource Allocator*. *Crisis Resolver* also produces crisis action to the network which will be consumed by other components in the architecture.

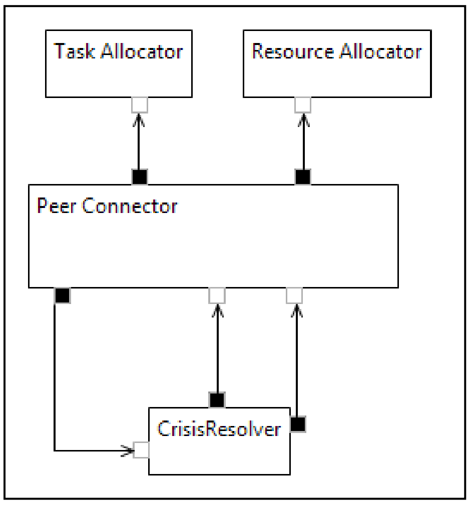


Figure 4.38. Visual peer-to-peer view

4.3.2.6. SOA Viewpoint

Service-oriented architectures consist of a collection of distributed components that provide and/or consume services. In SOA, service provider components and service consumer components can use different implementation languages and platforms. Services are largely standalone: service providers and service consumers are usually deployed independently, and often belong to different systems or even different organizations. The main benefit and the major driver of SOA is interoperability. SOA viewpoint provides guidelines for defining views for service oriented architectures [7].

We have defined DSL for SOA viewpoint and the metamodel for it is given below.

A SOA view model consists of *Service Providers*, *Service Consumers*, *Service Bus*, *Service Manager* and *Service Registry* components and their attachments. Service providers and consumers are possibly different applications, service registry is a persistent object and service manager is a process. Each of the components directly attached to service bus and communicates to each other through it. Each service bus must be definitely attached to a service registry and service manager. The service providers and service consumers are attached to the bus via their ports. Since the grammar is very complex and crowded, we cannot provide the grammar and abstract syntax for SOA viewpoint here due to space limitations.

An example visual SOA view for CMS application is given in Figure 4.39. *CMS Service Bus* enables CMS application to consume services of outsource applications such as *State Registry*, *Weather Forecasting Application*, *Traffic Monitoring System*, *Map Services*.

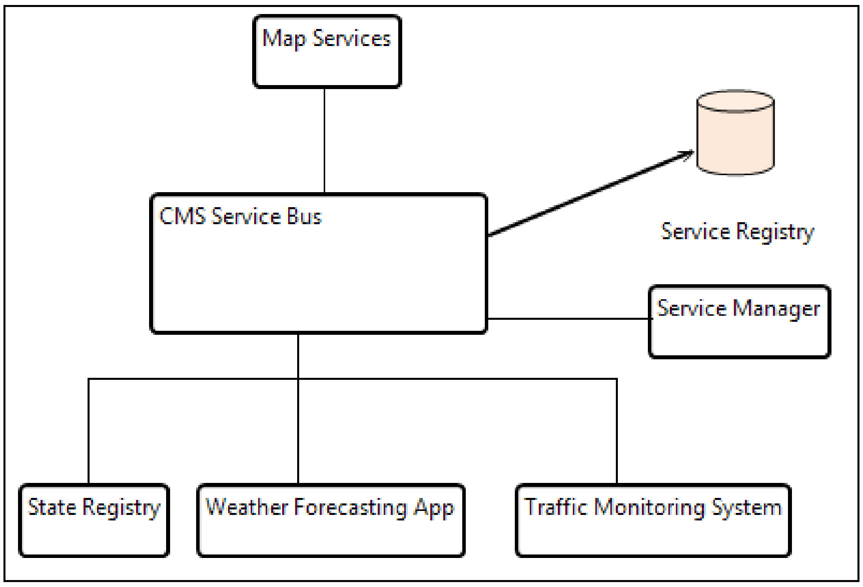


Figure 4.39. Visual SOA view

4.3.3. Allocation Viewpoints

4.3.3.1. Deployment Viewpoint

Deployment viewpoint, which is a kind of allocation viewpoint that is used to show how the software elements are allocated to hardware of a computing platform [7]. It is useful for analyzing and tuning certain quality attributes of the system such as performance, reliability and security. The DSL we have defined for V&B deployment style is presented in following sections.

The abstract syntax defined for deployment style is shown in Figure 4.40. The abstract syntax describes the elements of the language, which are *Software elements* and *Hardware elements*. The software elements are statically allocated to hardware elements by allocated to relation. In abstract syntax definition, we do not explicitly show this relation. It is implicit in the aggregation relation between hardware element and software element. The allocation of software to hardware does not have to be static. *Migration* relations are defined to support dynamic allocation schemes. There are three types of migration relations: *migrates to*, *copy migrates to*, *execution migrates to*. In addition to these viewpoint specific elements and relations, in order to reflect the topology of the platform *Connection* links between hardware elements are required.

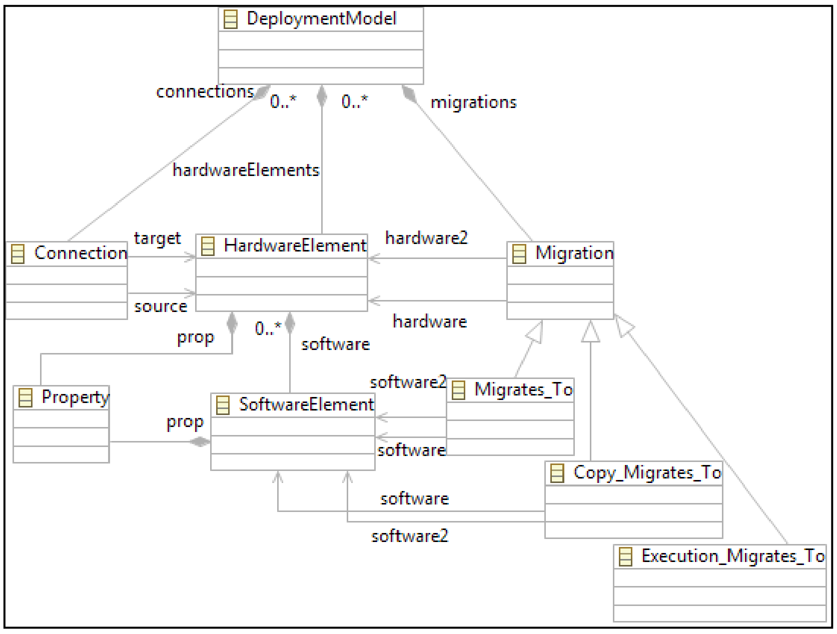


Figure 4.40. Abstract syntax for deployment viewpoint

The grammar for deployment style is also provided in Figure 4.41.

```
Model:{Model}
    (hardwareElements+=HardwareElement)*
    ('Connections' '{' (connections+=Connection)* '}')?
    ('Migrations' '{' (migrations+=Migration)* '}')?;

SoftwareElement: 'software' name=ID ':' type=ID ('{' (prop+=Property)* '}')?;

HardwareElement: 'HardwareElement' name=ID ':' type=ID
    ('{'((prop+=Property)* ( software+=SoftwareElement) ';')*'}')?;

Migration: Migrates_To | Copy_Migrates_To | Execution_Migrates_To;

Migrates_To:
hardware=[HardwareElement]'.'software=[SoftwareElement]
    'migrates_to' hardware2=[HardwareElement]'.'software2=[SoftwareElement]';';

Copy_Migrates_To: hardware=[HardwareElement]'.'software=[SoftwareElement]
    'copy_migrates_to' hardware2=[HardwareElement]'.'software2=[SoftwareElement]';';

Execution_Migrates_To: hardware=[HardwareElement]
    'execution_migrates_to' hardware2=[HardwareElement] ';';

Connection: source=[HardwareElement]
    'connected to' target=[HardwareElement] ('info' ':' connectionInfo=STRING)? ';';

Property: 'property' field=ID ':' value=STRING ';';
```

Figure 4.41. Grammar for deployment viewpoint

An example deployment view specified using both textual and visual concrete syntax for CMS architecture is provided in Figure 4.42. The visual concrete syntax defined for deployment view models software and hardware elements as elements, migrations and connections as relations.

```
HardwareElement CMS_Central_Machine: server {
    software CrisisHandler : process;
    software TaskAllocator : process;
    software ResourceAllocator : process;
}

HardwareElement CMS_Client: server {
    software CrisisWatcher : process;
    software CrisisReporter : process;
}

Connections{
    CMS_Client connected to CMS_Central_Machine;
}

Migrations {
    CMS_Client execution_migrates_to CMS_Central_Machine;
}
```

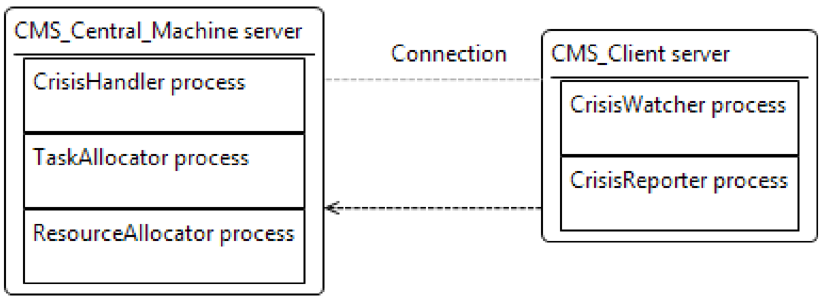


Figure 4.42. Textual and visual deployment views

We have identified four well-formedness rules for deployment style and implemented these as validation code. These rules are: (1) Every hardware element must be connected to at least one other hardware element. (2) An element cannot connect to itself (3) All types of migration relations have to be between two distinct hardware elements. (4) The source and target software element names referenced in migrates to and copy migrates to relations must be the same (i.e. the same software migrates from one hardware element to another).

4.3.3.2. Install Viewpoint

The install viewpoint allocates components of a C&C viewpoint to a file management system in the production environment. It helps to describe what specific files should be used and how they should be configured and packaged to deploy the system in a new environment. In addition it forms a guide for developers, deployers, and operators in order them to carry out their tasks properly.

The abstract syntax and the grammar for install viewpoint are given in Figure 4.43. The root element of install view model is *Model*. A model consists of *Directories*. Each directory consists of other *directories*, *files* or *components*. The components are run-time software elements. Directories, files and software components can have various properties.

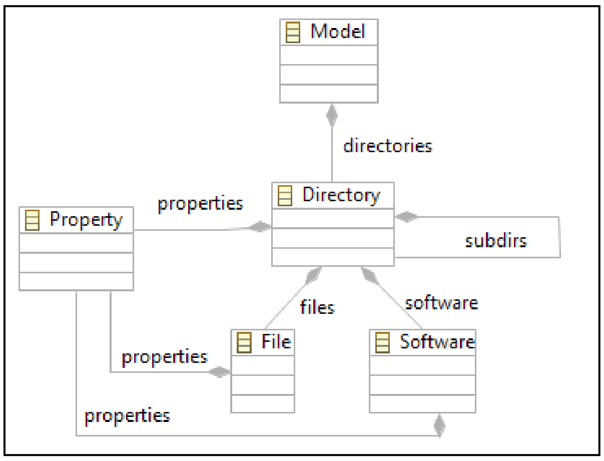


Figure 4.43. Abstract Syntax for install viewpoint

```
Model:
    (directories += Directory)*;

Directory:  'directory' name=ID (('{' (properties+=Property )*
        (subdirs+=Directory)* (files+=File)* (software+=Software)*'}') | ';');

File: 'file' name=ID
    (('{' (properties+=Property )* '}') | ';');

Software: 'software' name=ID
    (('{' (properties+=Property )* '}') | ';');

Property: 'property' name=ID '=' value=STRING ';';
```

Figure 4.44. Grammar for install viewpoint

Figure 4.45 shows an example visual install view of CMS system. The root directory is *CrisisManagementSystem*. Various directories are nested in it recursively. In the root directory *CMS.conf* file exists which contains configuration properties for crisis management system which can be edited at run-time. *Reporting* subdirectory includes crisis reports in it. *CrisisHandling* directory contains *CrisisListener* and *CrisisResolver* jobs in it.

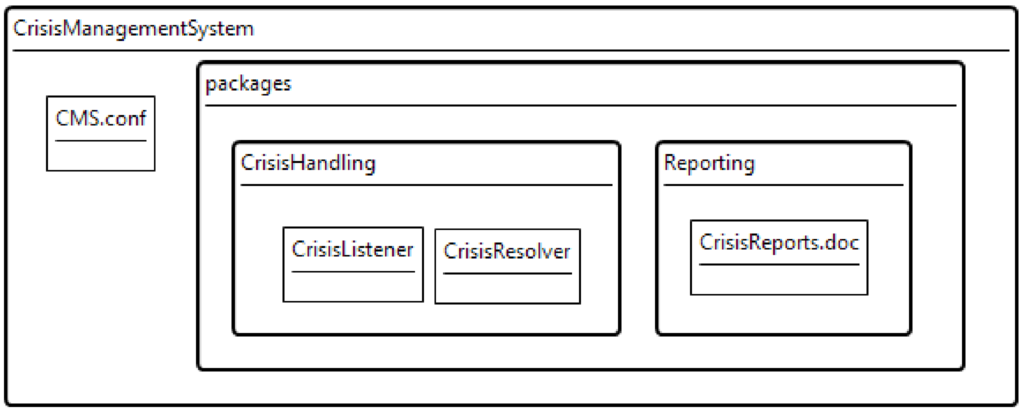


Figure 4.45. Visual install view

There are no additional topological constraints for install viewpoint except for all the directories, files and software components must be organized in a tree structure. We didn't need to implement any well-formedness rule for this because this constraint is already satisfied by our grammar definition.

4.3.3.3. Work Assignment Viewpoint

The work assignment viewpoint allocates modules of a module viewpoint to the groups and individuals who are responsible for the realization of a system. It defines the responsibility for implementing and integrating the modules to the appropriate development teams.

The abstract syntax and the grammar for work assignment viewpoint is shown in Figure 4.46, 4.47. The root element of the grammar is *Model*. A work assignment model consists of *Software elements* and *Environmental elements*. A software element can be a *Module* or *Subsystem*. An environmental element can be a

Person, *Team* or *Department*. Software elements are assigned to environmental elements via responsible for relation. Every element in the model can have various properties. The name and value of the property must be specified by the view definer.

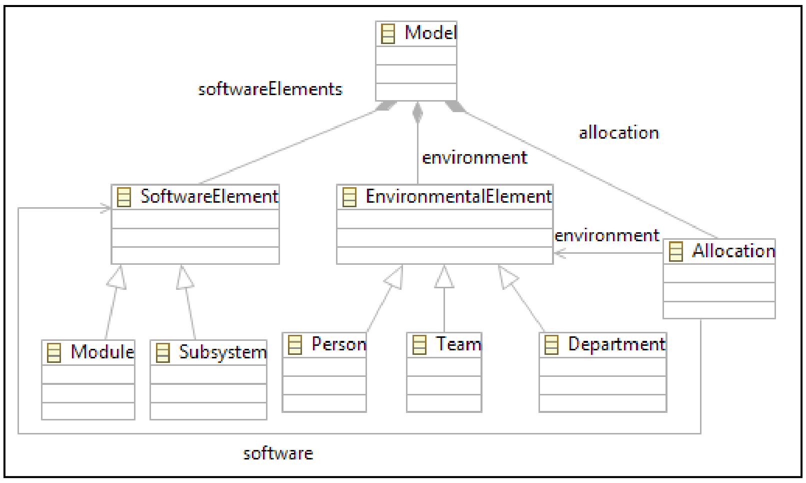


Figure 4.46. Abstract syntax for work allocation view

```
Model: {Model}
    (softwareElements+=SoftwareElement)*
    (environment+=EnvironmentalElement)*
    ('Allocations' '{' (allocation+=Allocation)*
    '}')?;

SoftwareElement: Module | Subsystem;

Module: 'module' name=ID ';';
Subsystem: 'subsystem' name=ID ';';

EnvironmentalElement: Person | Team | Department;

Person: 'Person' name=ID ';';
Team: 'Team' name=ID ';';
Department: 'Department' name=ID ';';

Allocation: software=[SoftwareElement] '->' environment=[EnvironmentalElement] ';';
```

Figure 4.47. Grammar for work allocation view

4.4. Evaluation of Architectural Viewpoint Frameworks

Architectural views represent an important input for defining the documentation of the architecture that consists of the description of multiple views and information beyond views. Documenting the architecture using architectural views helps improve modeling and likewise the early review of the system during architectural analysis. Yet, it appears that this review process has been basically at the level of architectural views and the evaluation of viewpoints has not been considered. However, if the architectural viewpoints are not well-defined then implicitly this will have an impact on the quality of the views and likewise the documentation of the architecture. We present an evaluation framework for assessing architectural viewpoints based on software language engineering techniques. The approach does not assume a particular architecture framework and can be applied to existing viewpoints or newly defined viewpoints. We illustrate our approach for reviewing viewpoints of the Views and Beyond approach.

4.4.1. Evaluation Framework

We provide an assessment framework for evaluating existing or newly defined architectural viewpoints. Our basic premise is that viewpoints can be considered as domain specific languages and likewise the evaluation of the viewpoint also considers the language aspects of the viewpoint.

Given the elements of a language we can now evaluate viewpoints, the ‘languages’ for defining views. A coarse-grained evaluation would be to check whether the language elements of abstract syntax, static semantics and concrete semantics, are defined for the viewpoints. This does not really provide much information since all the viewpoints seem to somehow describe the above elements albeit in a different degree, and as such the architectural viewpoint evaluation would not be of less practical value. To be able to refine the degree to which each element is addressed we propose to model each viewpoint explicitly as a domain specific language (DSL). The overall process for evaluating an architectural framework consisting of different viewpoint is shown in Figure 4.48.

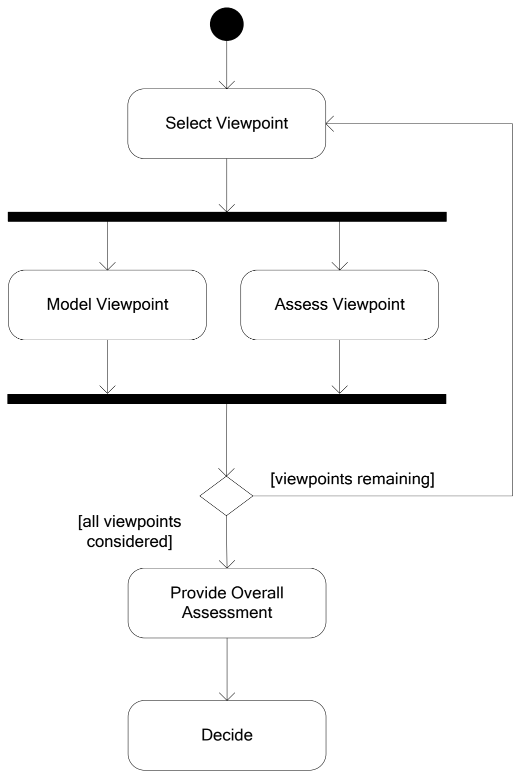
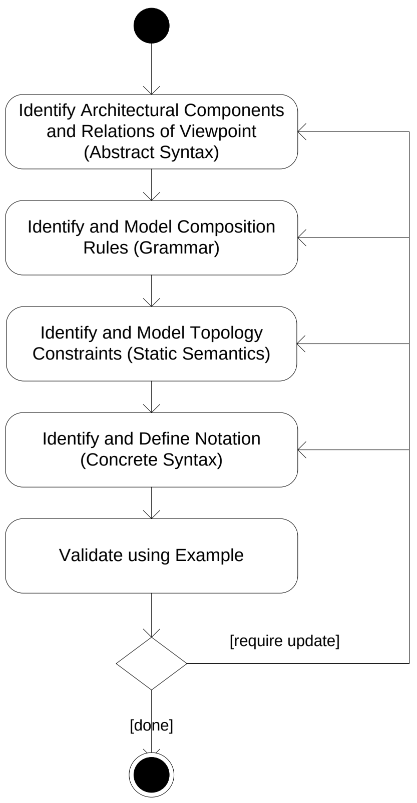


Figure 4.48. Overall Process for Assessment of Architectural Viewpoint

After selecting an architectural viewpoint, the viewpoint is modeled and in parallel the assessment of the corresponding viewpoint takes place. After all the viewpoints have been modeled and assessed, the overall assessment for the architectural framework is provided. Based on the overall assessment of the viewpoint(s) it is decided on what actions to take. Let's discuss each of these activities in more detail.

The activity Select Viewpoint selects a viewpoint that is provided either by a given architecture framework, or that has been newly introduced by viewpoint designers. Note that with this activity we mean the selection for evaluation of the viewpoint. Alternatively, required viewpoints are selected to derive architectural views based on the viewpoint. This is, for example, described by Clements et al. [7] in which viewpoints are selected with respect to the needs of stakeholders, available budget, the schedule, and the available skills.

The activity Model Viewpoints defines the DSL for the selected viewpoint and the detailed steps for this are shown in Figure 4.49. We have already explained this process in detail throughout chapter 4. For this, the description of the viewpoint in the literature (e.g. textbook) is analyzed.



Identify Architectural Components and Relations of Viewpoint (Abstract Syntax)
Identify and Model Composition Rules (Grammar)
Identify and Model Topology Constraints (Static Semantics)
Identify and Define Notation (Concrete Syntax)
Validate using Example
[require update]
[done]

Figure 4.49. Activity Diagram for Activity Model Viewpoint

The first step in the activity Model Viewpoints is the identification and definition of the architectural component and relation types. This is necessary to define the abstract syntax of the viewpoint. As stated before, the abstract syntax defines both the concepts (architectural component and relation types) of the language and the relations among these concepts. To represent the abstract syntax either a model-based approach or a grammar-based approach is adopted [23][35]. In the model-based approach, typically a UML model is provided defining the language concepts and their relations. In the grammar-based approach a grammar (e.g. EBNF grammar) is defined. In our approach we provide both a UML model and an EBNF-based grammar of the viewpoint. The composition rules are identified in the activity Identify and Model Composition Rules.

After the abstract syntax and the corresponding grammar/model have been defined the topology constraints (i.e. static semantics) are identified and modeled. The next activity is to Identify and Define the Notation (Concrete Syntax). Finally, the activity Validate using Example aims to define example models using the modeled viewpoint. The outcome of this activity might require iterating to the previous activities.

In parallel with the execution of the activity Model Viewpoints, also an evaluation of the viewpoint is carried out (activity Assess Viewpoint as shown in Figure). For evaluating the viewpoint we focus in particular on the elements of abstract syntax, concrete syntax and static semantics. We adopt the evaluation framework as defined in Table 1.

Table 1. Assessment framework for evaluating Architectural Viewpoints

Evaluation Level	**Description**
L1	Not defined
L2	Incomplete, Informally defined
L3	Complete, Informally defined
L4	Incomplete, Formally defined
L5	Complete, formally defined

The table distinguishes among four levels L1 to L5 indicating the quality and completeness of the corresponding element. As it can be seen in the table, a lower quality indicates that the corresponding element has not been described (missing, not defined) whereas a higher value indicates that the given element is completely defined and validated.

The activity Provide Overall Assessment in Figure 4.48 defines the summary of the overall evaluations of the viewpoints for the given architecture framework or set of viewpoints.

The final activity Decide in Figure 4.48 describes the recommendations and decisions on the usage of the selected viewpoints. In case the selected viewpoint is well-defined typically no action will be undertaken and the viewpoint can be used as is. If the viewpoint is not well-defined one may decide to enhance the viewpoint of the original viewpoint description. In that case, the assessment level (L1 to L5) will increase as well.

4.4.2. Evaluation of Views and Beyond Framework

Throughout Chapter 4, we have provided detailed explanation of meta-models created for Views and Beyond architecture viewpoint framework together with evaluation of their original viewpoint specifications from meta-modeling perspective. We have implemented all architectural viewpoints of V&B framework as domain specific languages and we state that the mapping of each viewpoint and its discussion is interesting by itself. The adopted approach was similar as defined in the previous section. As stated before, the domain specific language engineering approach has two benefits: (1) it helps to make the viewpoints executable (2) it provides insight in the degree of precision of the analyzed viewpoints.

In this section, we present an overall summary of our experience in mapping V&B architectural viewpoints to domain specific languages. For this we will use again our meta-model evaluation framework as we have defined in Table 1. We have applied the framework on each viewpoint defined by V&B. The viewpoints to evaluate are collected from both the first edition [6] and the second edition of the V&B book [7]. The evaluation results are presented in dot charts.

There are five levels in meta-model evaluation framework. The abstract syntax specifications in V&B framework do not exceed level L3. For both the editions of the book, the abstract syntax definition is in L3 for most of the viewpoints. This means that, no metamodel or grammar is provided for defining the abstract syntax. However, a clear textual specification is provided that can be easily mapped to model.

Figure 4.50 shows a dot chart that compares the degree of language precision of viewpoints in both editions of the V&B approach. With respect to the abstract syntax we can conclude that there is not much deviation between two editions of the book. Aspects, Data Model and SOA viewpoint values are in L1 for the first edition of the book, because those viewpoints are later introduced in the second edition. The same situation also applies to the communicating processes viewpoint for the second edition of the book, since it is excluded in the second edition. For most of the remaining viewpoints, abstract syntax definition levels overlap for both editions of the book. For generalization and publish-subscribe viewpoints a more clear textual description is provided in the second edition.

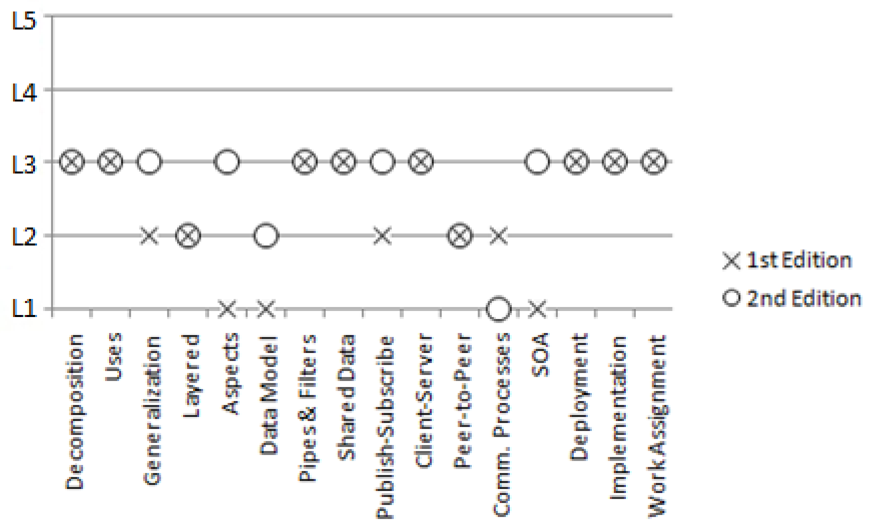


Figure 4.50. Abstract syntax definition levels for V&B (both editions of the book)

When we consider the concrete syntax definitions the deviation between two editions of the book is higher. For the module viewpoints (i.e. the first 6 viewpoints of the chart in Figure 4.51), the concrete syntax definitions are mostly in level L4, indicating that there is semi-formal concrete syntax definition for those viewpoints in both editions of the book. Mostly, UML is recommended as modeling notation explicitly showing how to use UML while realizing views for module viewpoints. For component-and-connector viewpoints (i.e. from 7th viewpoint to 13th viewpoint), the second edition of the book is still at L4. However, in the first edition of the book most of the C&C viewpoints are in L3-informal concrete syntax level. In the first edition, UML is mentioned roughly for the overall C&C viewpoints, however, it is not depicted how to use them for the specific viewpoints. In the second book, UML discussion for C&C viewpoints is again done for all viewpoints together, however, this time the discussion is detailed enough to specify how to use UML notations required for each viewpoint. For none of the viewpoints of the two editions, L5-formal concrete syntax level is reached. Although some formal modeling techniques such as ADLs are mentioned, it is not described how to use those ADLs for modeling with specific viewpoints.

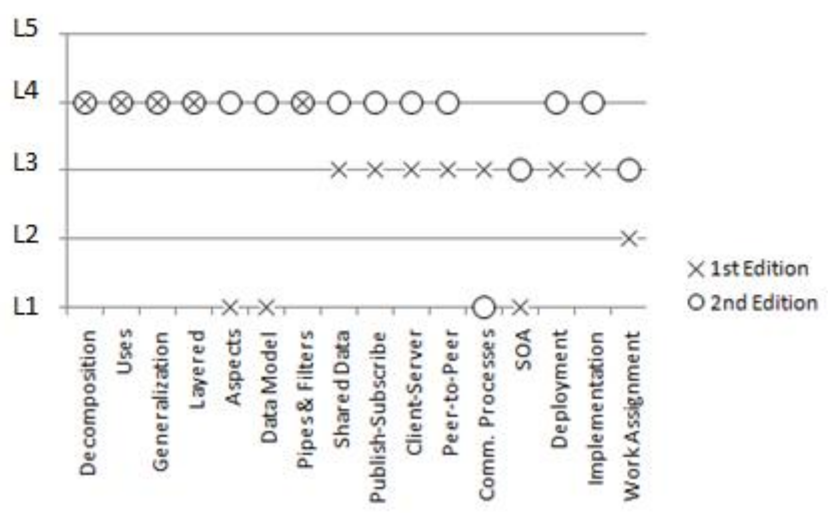


Figure 4.51. Concrete syntax definition levels for V&B (both editions of the book)

The static semantics definition for no viewpoint exceeds level 3-complete constraints in natural language. The constraints are always defined in natural language. There is some refinement of the constraint definitions in the second edition compared to those described in the first edition. In the first edition, 11 viewpoints are in L2 and L3 meaning that no constraints are specified or they are incomplete. In the second edition, four of those moves to L4 (uses, generalization, pipes&filters and publish-subscribe) meaning that they are still in natural language form however the constraints on language constructs are completely specified.

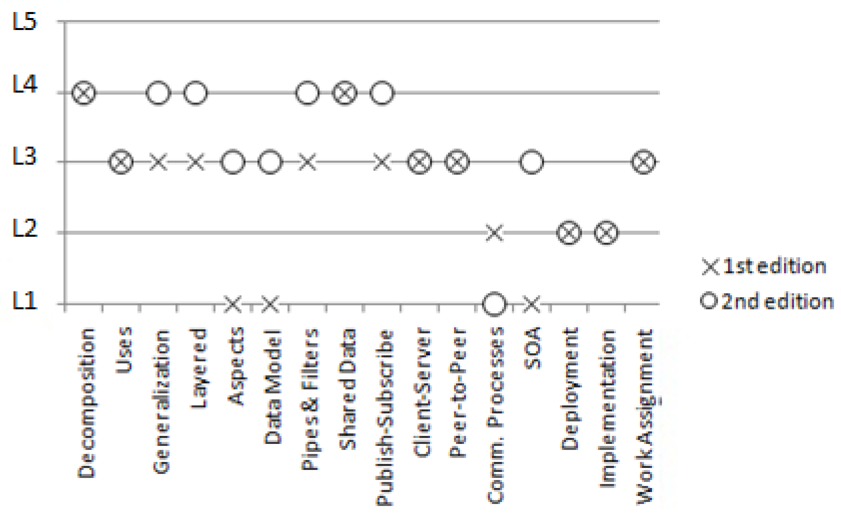


Figure 4.52. Static semantics definition levels for V&B

(both editions of the book)

We can conclude from this analysis that abstract syntax definition for V&B viewpoints are mostly in L3 and that these can be easily mapped to validated models as we do while defining DSLs. The concrete syntax definitions are mostly in L4. Informal and semi-formal notations are introduced and their usage is properly explained. However, no formal notations are provided. The constraints on viewpoint elements and relations are always provided in natural language form. This informality causes incomplete specification of constraints: there are only few viewpoints in L4. By defining DSLs for V&B approach, we have made the viewpoint definitions in L5 for each category: abstract syntax, concrete syntax and static semantics.

Chapter 5

SAVE-Bench Tool

We have combined the DSLs we have defined for viewpoints in our tool SAVE(Software Architecture View Modeling Environment)-Bench. Save-Bench is an architectural modeling environment that enables architecture stakeholders to develop textual and visual view models. Save-Bench is implemented using various MDE tools and published as an Eclipse plug-in. It is open to new domain specific language(viewpoint) additions.

Throughout the thesis, we have explained how we defined viewpoints as domain-specific languages in an abstract way. In this section, we first explain Save-Bench architecture from the point of view of language developer. We present the MDSD tools we have used and how we used them to define DSLs. Then, we explain Save-Bench, from the user (view modeler) point of view. We describe how a view modeler can use Save-Bench.

5.1. Save-Bench Architecture

Tool support has been found essential in the comparison of ADLs [27]. This holds also for mapping viewpoints to DSLs. We have implemented the domain specific languages for all the 15 architectural viewpoints in an Eclipse plug-in tool that we call Software Architecture Environment for Modeling Views (SAVE-Bench). SAVE-Bench enables the creation of architecture projects and the modeling of architectural views based on the defined viewpoints. In the following we describe the architecture of Save-Bench toolset together with how to extend the tool with a new viewpoint DSL.

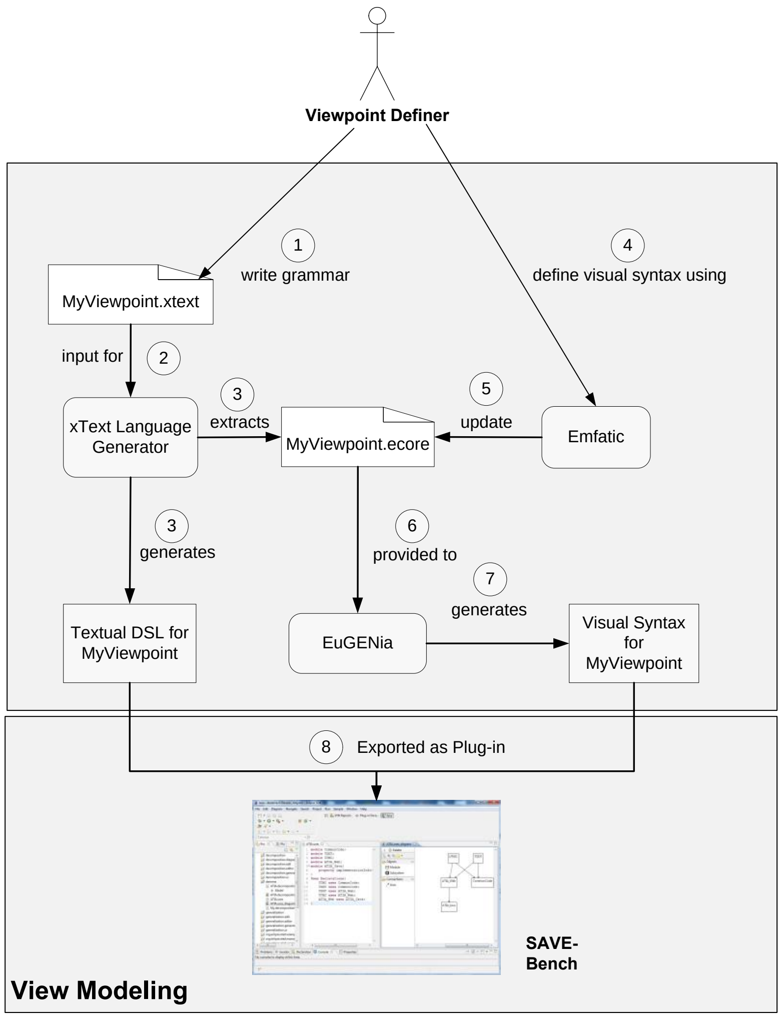


Figure 5.1. Process for defining Viewpoints as DSLs and generating SAVE-Bench

Figure 5.1 shows the process of defining example viewpoint MyViewpoint as a domain specific language and exporting it as Eclipse plug-in. Various tools such as Xtext [39], GMF [16], EuGENia [13] and EMFatic [12] are used in the language definition process. Their roles will be explained in the following.

Firstly, the viewpoint definer creates an empty Xtext project and writes grammar for MyViewpoint. The grammar definition is written into the file with .xtext extension using the Xtext editor and following the rules of Xtext's EBNF like grammar definition language. Xtext [39] is a part of Eclipse TMF (Textual Modeling Framework) project and it enables creation of domain specific languages from grammar definitions. After writing the grammar, Xtext language generator is run and it builds full implementation of the domain specific language for the written grammar. The DSL implementation runs on Java virtual machine. After generation, the DSL implementation is ready to be exported as plug-in to Eclipse. If required, the generated Java code may be modified before exporting. For example, we modify it in order to add well-formedness rules to our language model. The language generator generates empty validation class for a model. We implement validation class for a viewpoint with the constraints of that viewpoint that cannot be satisfied by metamodel definition.

The grammar definition is in fact a metamodel definition where textual concrete syntax is embedded in it. The Xtext language generator extracts the metamodel from grammar and outputs it as Ecore metamodel. We use this Ecore metamodel as the abstract syntax definition while defining visual concrete syntax for our DSL.

In order to explain the process of defining visual concrete syntax for our DSLs, we need to introduce some graphical modeling tools. Traditionally, GMF (Graphical Modeling Framework) tools [16] are used in order to define visual concrete syntax based on an Ecore metamodel. GMF tools provide a set of generative components for generating diagram editors in Eclipse. However, the process of defining a visual editor using GMF is a bit complex and requires knowledge of low level details related to editor. That is why; we use the tool EuGENia [12] that is introduced in order to raise the level of abstraction in GMF. It automatically generates required models for GMF diagram generation from a single annotated Ecore metamodel. In order to be able to annotate the Ecore metamodel with visual concrete syntax information, we utilize EMFatic.

With the help of EMFatic tool, the viewpoint definer annotates the Ecore metamodel with visual concrete syntax information. This annotation is done using a special language provided by EuGENia. Shortly, in the annotation process, the viewpoint definer states which elements of metamodel are represented by which graphical notations. The resulting Ecore metamodel is given as an input to EuGENia generator. Eugenia generates required models for GMF diagram editor generation. From those models, GMF diagram editor is directly generated.

Lastly, both textual and visual editors defined for viewpoint are exported as plug-ins to Eclipse. A view modeler can use those editors to model architecture views based on the viewpoint. Both visual and textual models run on the same model instance, that is why, a change to one of those models affects both models.

5.2. Using Save-Bench

As described in the previous section the Save-Bench tool runs on the Eclipse Platform. It consists of a set of predefined DSLs for modeling architectural views. The tool provides visual and textual editors that runs based on those DSLs. In this section, we describe the tool and show how it can be used to model architectural views.

In order to use the languages provided by Save-Bench, the view modeler must first open an empty project. Then, the desired view model files can be created by using the file creation wizard. A sample view from Eclipse's file creation wizard showing the SAVE category that we have added is shown in Figure 5.2. In order to create architectural views using textual concrete syntax, the model files must be created. In order to use visual concrete syntax, diagram files must be created.

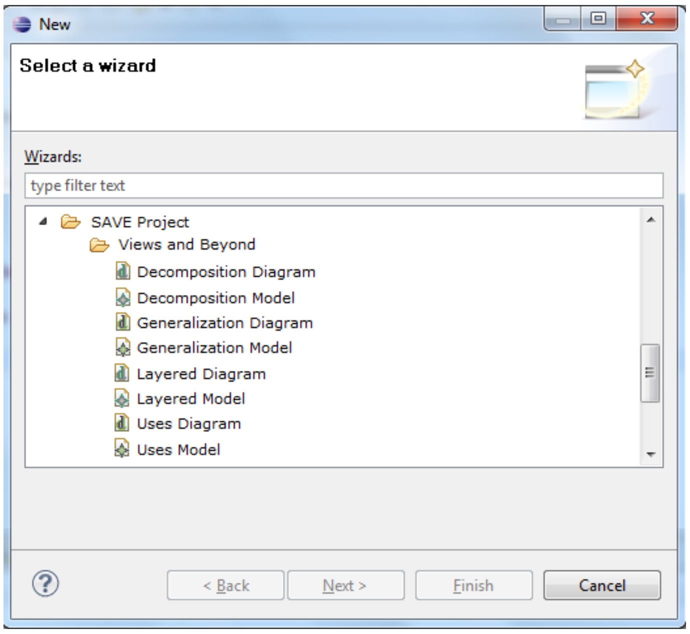
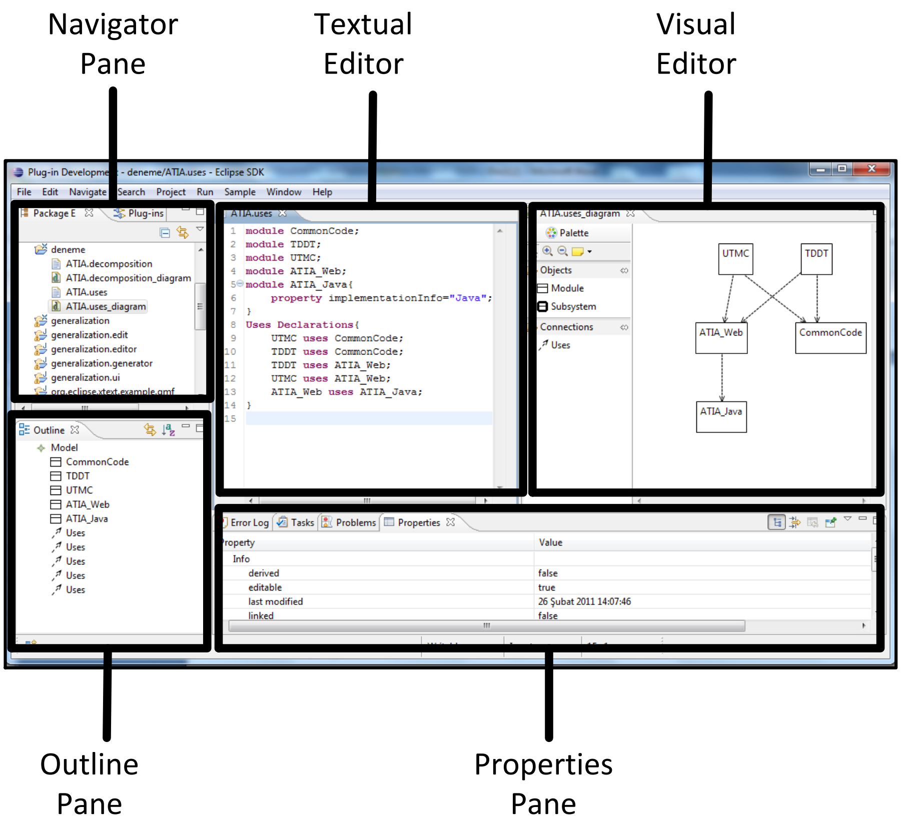


Figure 5.2. SAVE-Bench model file creation wizard

Figure 5.3 shows a sample screenshot from the SAVE-Bench tool. SAVE-Bench provides a user interface with 5 different panes; 1) Navigator Pane (top-left), 2) Outline Pane (bottom-left), 3) Properties Pane (bottom-right), 4) Textual Editor Pane (middle), and 5) Visual Editor Pane (right). Navigator Pane is for managing view models. All of the views created for the architecture are listed there. The views can be opened or new views can be created using Navigator Pane. Outline Pane shows the current structure of selected view as a list. Properties Pane is useful when using visual editor and it enables the view modeler to modify the properties of elements for which visual concrete syntax is not defined.

The textual editor pane enables modeling the view by writing model code. The language for model code was previously defined by grammar of DSL for that view. Help documentation is provided for each view, in order to explain the syntactic rules to be followed while modeling that view textually. The textual editor eases the view modeler's task by providing highlighting and auto-completion functionality. The keywords specified for a view such as module are highlighted by textual editor. Also, while writing the model code, the auto-completer guides the view modeler.



```
module CommonCode;
module TDDT;
module UTMC;
module ATIA_Web;
module ATIA_Java{
    property implementationInfo="Java";
}
Uses Declarations{
    UTMC uses CommonCode;
    TDDT uses CommonCode;
    TDDT uses ATIA_Web;
    UTMC uses ATIA_Web;
    ATIA_Web uses ATIA_Java;
}
```

roperty	Value
Info	
derived	false
editable	true
last modified	26 Şubat 2011 14:07:46
linked	false

Figure 5.3. Snapshot of the SAVE-Bench tool for modeling architectural views

The visual editor pane enables modeling the view by using diagram components in a drag-and-drop manner using the provided tool palette. When a model is created using visual editor, the textual model is automatically created for that. The visual editor is useful for view modeler’s that are not familiar with the syntactic rules and constraints imposed by textual editor. Visual concrete syntax is mostly developed for only elements and relations; it does not support to manage details of them. The properties pane is useful for defining those details (e.g. interface visibility property of a module) for which visual concrete syntax is not defined. The visual and textual models runs on the same virtual model, that is why, when any one of them are changed, the other is also affected by that change.

Project Creation: The tool can be used to document the views for an architecture for a given software project. An Eclipse workbench is started in which the Save-Bench plug-in is already installed. The tool offers a so-called project wizard in which the project details such as, the name and location of the project is specified. In addition, first one of the available architecture framework approaches (e.g.

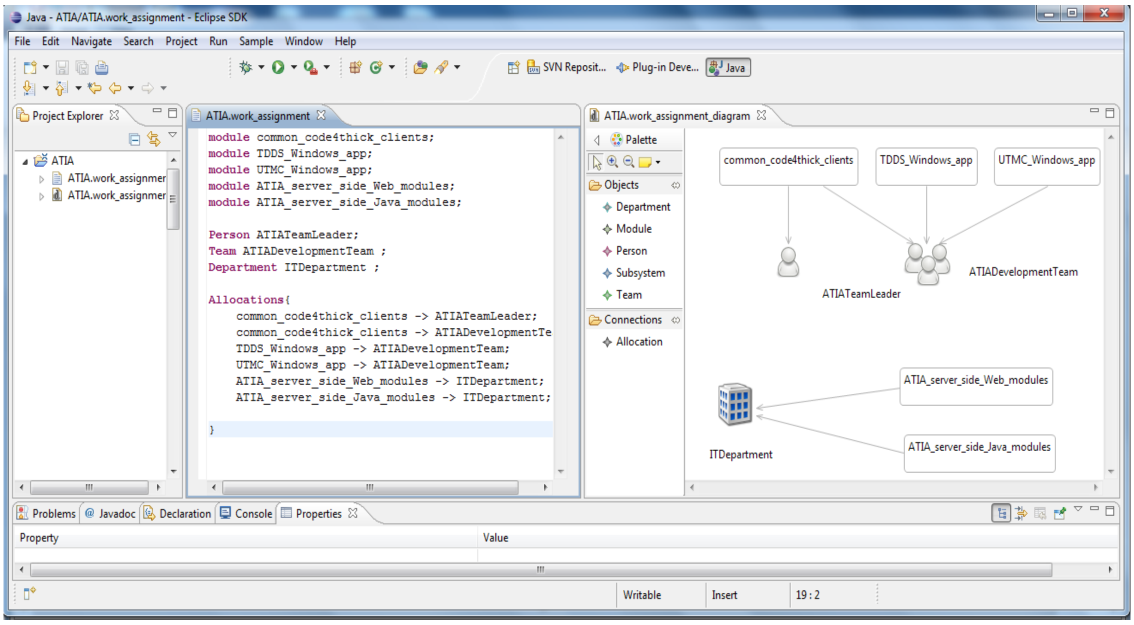
V&B) is selected, and then the viewpoints that are necessary for the project. The selection of the viewpoints can be carried out as defined by the architectural viewpoint selection process in [7]. The tool also provides means to define the details of the stakeholders and their interest for the particular views.

Definition of Textual Architectural Views: The textual model for a view is constructed obeying the syntactic rules that are defined by the corresponding DSL. An explanation of those rules for each view is provided in the help section menu prepared for Save-Bench project. When the syntactic rules are not satisfied in the designed model, errors and warnings are given.

Generation of Visual View: After defining the textual view model, the visual form of it can be generated. For this, first the textual model file needs to be selected from Project Explorer window. After which diagram creation wizard can be started. In the wizard, metadata for the diagram such as target folder and diagram root element is specified and the selected view is created and opened. The user can modify the view using either textual or visual model. Whenever one of these models is modified, the other one is automatically updated.

Definition of Visual Architectural View: Instead of generating visual model from textual model, the visual model can also be defined from scratch. Using new file creation wizard of Eclipse, the desired diagram type is selected. The architectural view is modeled as diagram using the "Palette". No generation is required for getting textual model from visual model. When visual model is created, the textual model is automatically created and updated according to modifications on visual model.

A screenshot from Save-Bench project is seen Figure 5.4. Work assignment view of architecture is defined textually and visually.



```
module common_code4thick_clients;
module TDDS_Windows_app;
module UTMC_Windows_app;
module ATIA_server_side_Web_modules;
module ATIA_server_side_Java_modules;

Person ATIATeamLeader;
Team ATIADevelopmentTeam ;
Department ITDepartment ;

Allocations{
    common_code4thick_clients -> ATIATeamLeader;
    common_code4thick_clients -> ATIADevelopmentTe
    TDDS_Windows_app -> ATIADevelopmentTeam;
    UTMC_Windows_app -> ATIADevelopmentTeam;
    ATIA_server_side_Web_modules -> ITDepartment;
    ATIA_server_side_Java_modules -> ITDepartment;

}
```

Figure 5.4. SAVE-Bench screenshot

Chapter 6

Automatic Architecture Document Generation

Defining DSLs for viewpoints is a way of formalizing the viewpoints. We can benefit from this formalization in various ways such as automatic model validation, model generation etc. In this section, we will explain how we used the DSLs for viewpoints to support automatic architecture documentation generation.

Every architecture needs a documentation to guide architecture stakeholders about how to benefit from the architecture and clarify ambiguous points. Architecture documentation is a communication artifact for all stakeholders and it is used during the whole lifecycle of the architecture. It contains both natural language descriptions about system and formal architecture models. We utilize our DSLs for architecture viewpoints in order to automatically generate the architecture view related part of the architecture documentation. The generation is done via Model-to-Text (M2T) transformation.

A model transformation takes as input a model that conforms to a given metamodel and produces as output another model that conforms to another metamodel. A M2T transformation is a special case of model transformation where target is just strings. We applied model-to-text transformation on the view models in order to generate architecture documentation. Figure 6.1 shows the process for generating architecture document from view models.

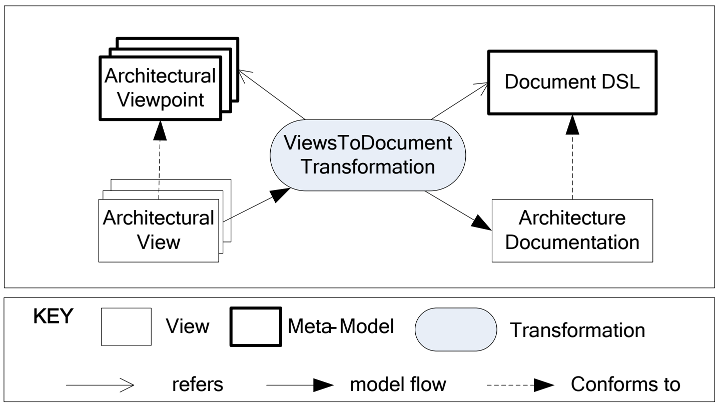


Figure 6.1. Model-to-text transformation pattern for architecture documentation generation

A set of architecture views is given as input to ViewsToDocument transformation engine. Each view conforms to its viewpoint. The engine transforms the required information to target strings and combines them into architecture documentation. The output model conforms to Document DSL which describes the organization of knowledge that is collected from architectural views. We decided to organize the knowledge as follows: A view is documented with rationale behind producing that view, the stakeholders interested in that view and an element catalog. Since our view models do not have rationale and stakeholder information, we left those fields blank in the generated document for the architect to fill in after generation is completed. An element catalog lists the elements used in the view together with natural language explanation and some selected important properties of elements.

In order to better explain the view to document transformation process, we present Figure 6.2. We need to have our view models at hand before we run transformation. The architect is responsible for defining the view models or ensuring they are already defined by the responsible stakeholders, selecting the viewpoint types to give as an input to transformation engine, running transformation and doing final revision on the generated architecture document. The transformation engine for each viewpoint is different, so, they are run separately. The output of a transformation is output text which is appended to the end of output document. After the document is generated the architect needs to do review the documentation to fill in blank parts such as rationale for view, stakeholders and view model figures etc. if required.

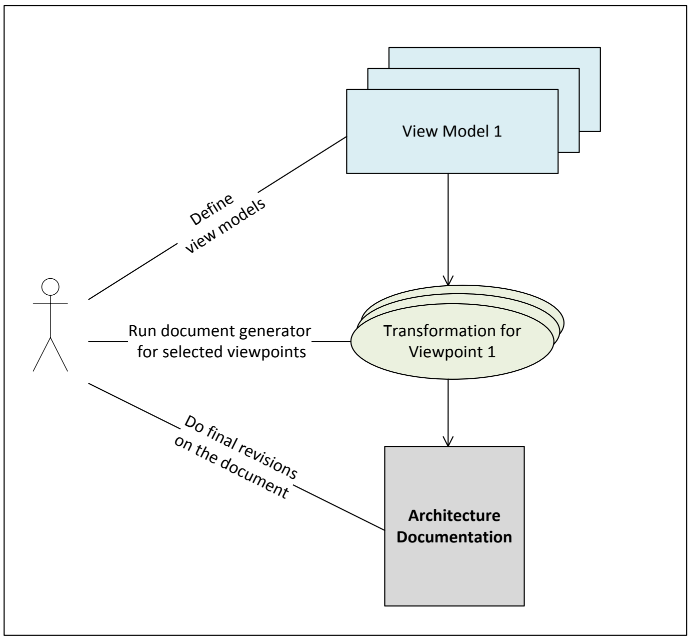
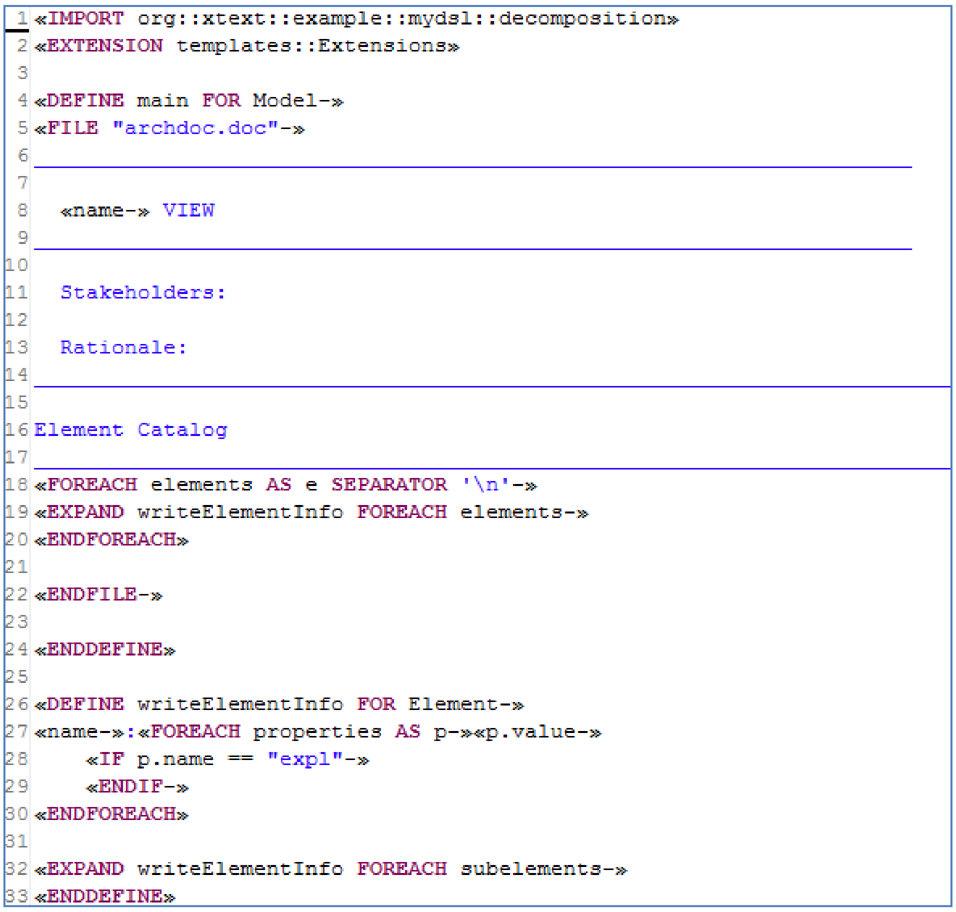

View Model 1
Define view models
Run document generator for selected viewpoints
Transformation for Viewpoint 1
Do final revisions on the document
Architecture Documentation

Figure 6.2. Architecture documentation generation process

So far, we have explained the overall process of generating architecture documentation from views. Here, we will go further to explain the tools and techniques we used when defining the architecture document generator. The transformation language we have used is Xpand [38] which is part of the Eclipse M2T project. It allows defining templates for transforming models to texts. An example transformation template that is written for decomposition view is shown in Figure 6.3.

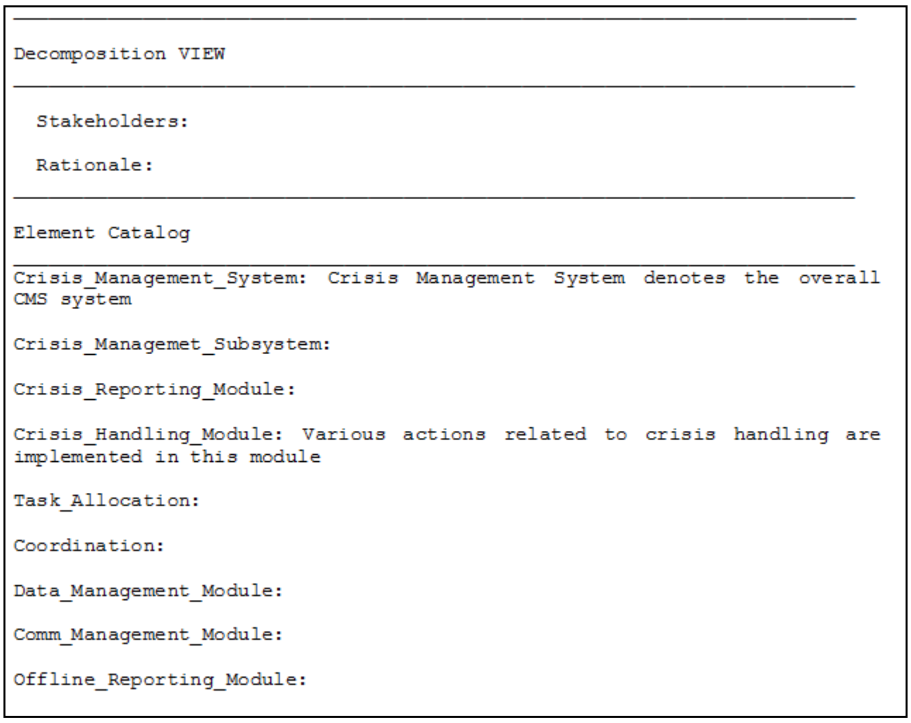


```
 1 «IMPORT org::xtext::example::mydsl::decomposition»
 2 «EXTENSION templates::Extensions»
 3
 4 «DEFINE main FOR Model-»
 5 «FILE "archdoc.doc"-»
 6
 7
 8   «name-» VIEW
 9
10
11   Stakeholders:
12
13   Rationale:
14
15
16 Element Catalog
17
18 «FOREACH elements AS e SEPARATOR '\n'-»
19 «EXPAND writeElementInfo FOREACH elements-»
20 «ENDFOREACH»
21
22 «ENDFILE-»
23
24 «ENDDEFINE»
25
26 «DEFINE writeElementInfo FOR Element-»
27 «name-»:«FOREACH properties AS p-»«p.value-»
28     «IF p.name == "expl"-»
29     «ENDIF-»
30 «ENDFOREACH»
31
32 «EXPAND writeElementInfo FOREACH subelements-»
33 «ENDDEFINE»
```

Figure 6.3. M2T transformation template for decomposition viewpoint

The first line shows that uses metamodel is imported. “main” template is defined for Model which denotes a decomposition view model. The output file is indicated on 5th line of the code as archdoc.doc. The strings that will be directly printed on the output document are freely written. The values of model elements are taken in «..» block. FOREACH blocks are used to traverse list of model elements. For example, the FOREACH block on 18th-20th line of the code traverses the elements of decomposition model and prints required model values such as element name, explanation. EXPAND statement calls the write element info block, which recursively traverses elements and subelements of the view model and writes information about them. The explanation of a view element is defined as property in the view model where we have reserved “expl” named property for this purpose. In the document generator for decomposition view, we only pass “expl” named properties to output document as element explanation. For different views, we sometimes needed to pass other selected properties to output document. The transformation definition for each viewpoint of Save-Bench is given in Appendix A.

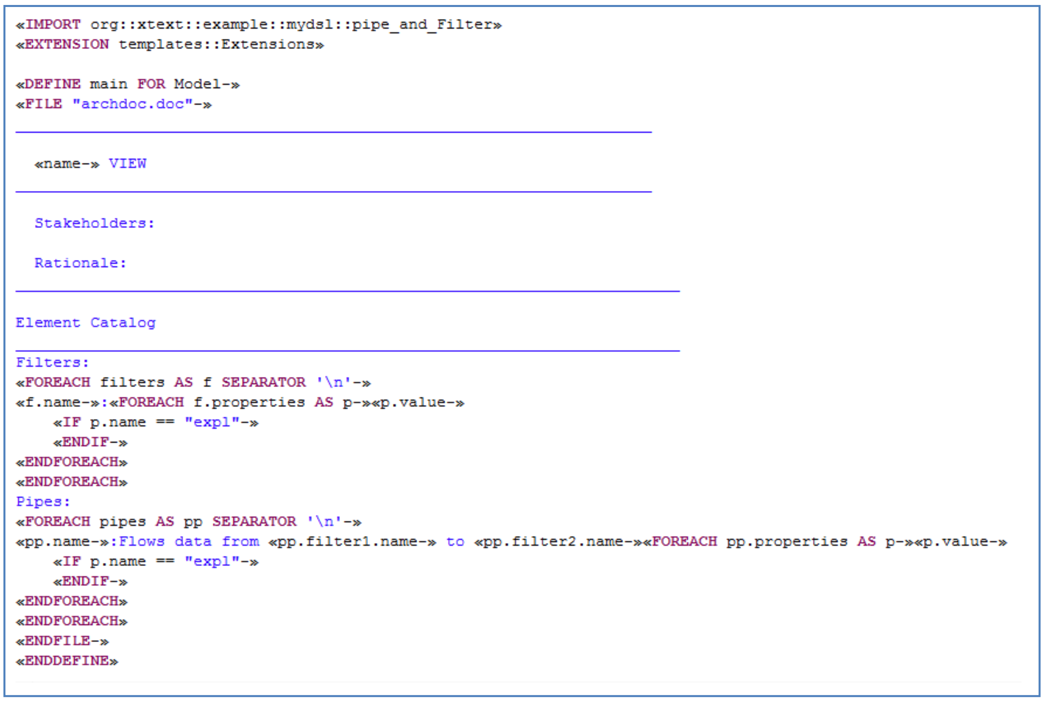
The decomposition view part of sample generated architecture documentation is shown in Figure 6.4.



```
________________________________________________________________________
Decomposition VIEW
________________________________________________________________________
  Stakeholders:

  Rationale:
________________________________________________________________________
Element Catalog
________________________________________________________________________
Crisis_Management_System:  Crisis  Management  System  denotes  the  overall
CMS system

Crisis_Managemet_Subsystem:

Crisis_Reporting_Module:

Crisis_Handling_Module:  Various  actions  related  to  crisis  handling  are
implemented in this module

Task_Allocation:

Coordination:

Data_Management_Module:

Comm_Management_Module:

Offline_Reporting_Module:
```

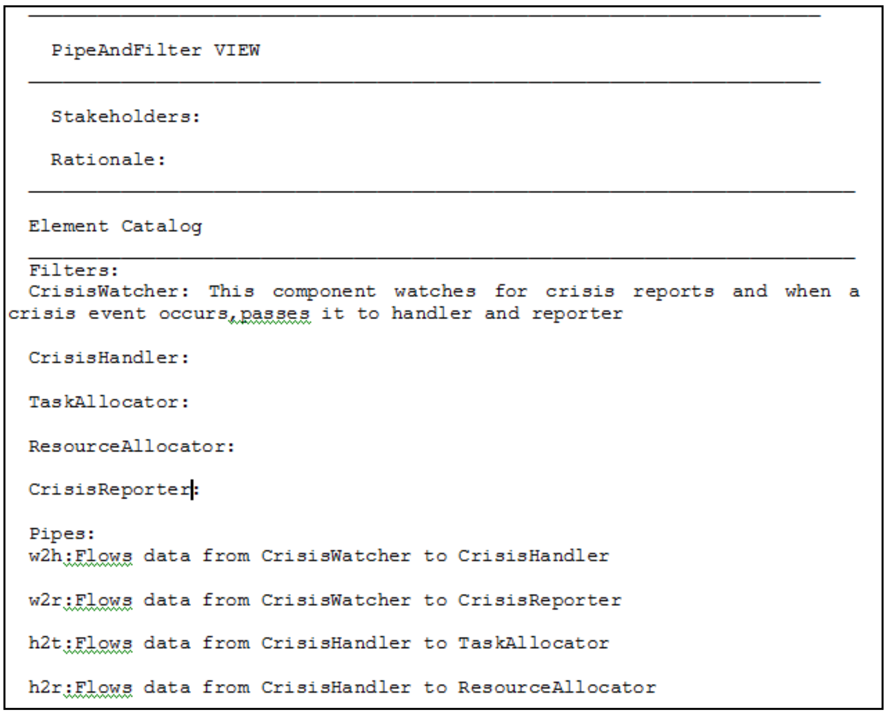
Figure 6.4. Decomposition view part of generated architecture documentation

Figure 6.5 shows the generation template for pipe and filter viewpoint. Here no expand statements are used recursively since there is not recursively nested elements in this model. Names and explanations of the filters are listed, and then the pipes are listed showing that from which filter they do flow data to which other filter. The pipes can also have explanations. Figure 6.6 shows the pipe and filter part of the generated output document.



```
«IMPORT org::xtext::example::mydsl::pipe_and_Filter»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
___________________________________________________________________

  «name-» VIEW
___________________________________________________________________

  Stakeholders:

  Rationale:
______________________________________________________________________

Element Catalog
______________________________________________________________________
Filters:
«FOREACH filters AS f SEPARATOR '\n'-»
«f.name-»:«FOREACH f.properties AS p-»«p.value-»
    «IF p.name == "expl"-»
    «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
Pipes:
«FOREACH pipes AS pp SEPARATOR '\n'-»
«pp.name-»:Flows data from «pp.filter1.name-» to «pp.filter2.name-»«FOREACH pp.properties AS p-»«p.value-»
    «IF p.name == "expl"-»
    «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Figure 6.5. M2T transformation template for pipe and filter viewpoint



```
  ____________________________________________________________________

    PipeAndFilter VIEW
  ____________________________________________________________________

    Stakeholders:

    Rationale:
  _______________________________________________________________________

  Element Catalog
  _______________________________________________________________________
  Filters:
  CrisisWatcher: This component watches for crisis reports and when a
crisis event occurs,passes it to handler and reporter

  CrisisHandler:

  TaskAllocator:

  ResourceAllocator:

  CrisisReporter:

  Pipes:
  w2h:Flows data from CrisisWatcher to CrisisHandler

  w2r:Flows data from CrisisWatcher to CrisisReporter

  h2t:Flows data from CrisisHandler to TaskAllocator

  h2r:Flows data from CrisisHandler to ResourceAllocator
```

Figure 6.6. Pipe and filter view part of generated architecture documentation

Chapter 7

Related Work

Architecture description languages (ADLs) have been proposed to model architectures. For a long time there have been little consensus on the key characteristics of an ADL. Different types of ADLs have also been introduced. Some ADLs have been defined to model a particular application domain, others are more general-purpose. Also the formal precision of the ADLs differ; some have a clear formal foundation while others have been less formal. Several researchers have attempted to provide clear guidelines for characterizing and distinguishing ADLs, by providing comparison and evaluation frameworks. Medvidovic and Taylor [27] have proposed a definition and a classification framework for ADL which states that an ADL must explicitly model components, connectors, and their configurations. Furthermore, they state that tool support for architecture-based development and evolution is needed. These four elements of an ADL include other sub-elements to characterize and compare ADLs. The focus in the framework is thus on architectural modeling features and tool support. In adopting a software language engineering approach we have focused on the three language elements of abstract syntax, concrete syntax and static semantics. In fact we could analyze also existing ADLs based on the approach in this thesis. That could be complementary to earlier evaluations of ADLs.

Model-driven development and language engineering concepts have had their impact on architectural modeling. ACME was one of the earlier ADLs that was developed as an architecture interchange language across ADLs . ACME has resulted from an analysis of notations for modeling architectures and was considered as the least common denominator of the existing ADLs. Because ADL would support the mapping of architectural specifications from one ADL to another, it would also enable integration of support tools across ADLs. Mapping between ADLs is a special case of model transformations as defined in the model-driven software development. A similar problem can occur in mappings of viewpoints of different viewpoint frameworks. Currently there is no explicit support for this in the literature. Our vision in this thesis is that once we have defined viewpoints as domain specific languages then this would also ease the mapping among different views. A viewpoint similar to ACME should then be defined using software language engineering techniques. We consider this as part of our future research.

xADL[18] has been introduced to support modularity and extensibility of architectural modeling. Despite earlier ADLs xADL is not a single fixed ADL but encapsulates various ADL features in modules that can be composed to form new ADLs. This is achieved by using the extension mechanisms provided by XML and XML schemas. xADL forms the basis for the ArchStudio 4 [18], an open-source software and systems architecture development environment including tools for modeling, visualizing, analyzing and implementing software and systems architectures. It is based on the Eclipse open development platform. Similar to our tool it is an architecture meta-modeling environment that can be used to define new views. In ArchStudio, new viewpoints could be defined by extending the core language. In our approach we focus on the software language engineering elements of abstract syntax, concrete syntax and static semantics. In addition viewpoints can be defined from scratch using Xtext or extended.

In addition to proper definition of viewpoints several authors have indicated the need for integration of viewpoints. For this the relations among architectural viewpoints and views need to be made explicit. A good overview and motivation for characterizing the relations among views is given by Boucké et al. [4]. They indicate that in the current literature on architecture view modeling relations among architectural views are not first-class abstractions. Based on a literature study Boucké et al. propose a framework for analyzing approaches to relations between views in three dimensions: usage, scope and mechanism. Each of the criteria focuses on a particular aspect of view relations. We believe that most of the issues addressed in [4] can be mainly achieved by adopting a software language engineering approach. In addition to specifying architectural views we could also define the relations among views. This is not addressed yet in our work but we will address this in the future.

Tekinerdogan et al. [36] has discussed the impact of evolution of concerns in architectural views. In case of evolution of the software system the related architectural views need to be adapted accordingly. To synchronize the architectural views it is necessary that the dependency links among the architectural concerns in the architectural views can be easily traced. They have documented explicit trace relations between architectural concerns, the architectural elements that address the concerns, and between architectural elements in general. In case of evolution of concerns one can follow the trace links to update and synchronize architectural views, keeping the software architecture consistent. In this thesis, we have now an approach for more formally specifying the viewpoints. Since architectural views can now be interpreted by the tool, we can now better provide support for automatic impact analysis over the architectural views and the (semi-) automatic update and synchronize architectural views.

So far, in the domain of software architecture the notion of architectural viewpoints has been basically viewed at the level of blueprints. Yet, in the enterprise architecture domain several authors have focused on the formalization of architectural viewpoints. Different attempts have been made before to model viewpoints as domain specific languages.

ArchiMate [2] is an enterprise architecture (EA) modeling language that is specified by concepts that focus on business, applications and technology domains. Those concepts form the base metamodel of ArchiMate language. A set of viewpoint languages are defined by composing the concepts available in the metamodel. Contrary to their approach, our viewpoint languages does not depend on a predefined set of concepts, each viewpoint has an independent language that defines its own concepts. This design choice makes it easy to introduce new viewpoints to the framework. However, it is impossible to define new viewpoints in ArchiMate if the required concepts are not available at the base metamodel. An additional extension mechanism is needed for this purpose [32].

Another example to attempts on formalizing EA viewpoints is about RM-ODP viewpoints. Vallecillo et al. initially focused on formally specifying the abstract languages provided by viewpoint specifications using a rewriting logic based framework Maude [10]. Later on, they also tackle the viewpoint formalization problem from model-driven development perspective and defined UML profile for viewpoints of RM-ODP [19] [33]. Lastly, they define textual notation for ODP specifications together with tool support [15]. The main difference of their approach and our study is the level of formality of the targeted viewpoint specifications. RM-ODP is specified by a standard that precisely defines the syntax and semantics of the language. So, the task of formalizing RM-ODP viewpoint specifications is transforming the present languages to executable languages and defining notations for using the language. However, in our work, we also address viewpoint specifications those are not specified precisely as languages. We offer software language engineering as a method for lifting existing viewpoint specifications to formal language level and provide a complete description of the method.

Chapter 8

Conclusions

The discipline of software architecture description has substantially evolved in the last decades. We can characterize the evolution from the following two perspectives.

First of all, the awareness that architecture should be modeled using multiple views. Having multiple views of the architecture helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. In the literature, initially views were not explicit, later a fixed set of viewpoints has been proposed to model and document the architecture. Because of the different concerns that need to be addressed for various systems, the current trend recognizes that the set of views should not be fixed but open-ended. The second dimension of evolution considers the formal precision of the architectural descriptions. Initially software architecture was represented using arbitrary box-and-lines notations leading to ambiguous interpretations. Later on, it was acknowledged to provide more formal support for architectural modeling, both visually and textually.

The work that we have presented in this thesis aims to elaborate on the evolution of these two dimensions. To provide an open ended-viewpoint approach in which viewpoints are formally specified we have stated that a software language engineering approach is necessary. The key premise behind this assumption is that viewpoints are in essence domain specific languages, and as such should be considered and developed like that. To validate our statement we have analyzed the viewpoints in the Views and Beyond approach, and defined all these viewpoints as domain specific languages. We have compared both the first edition and second edition of the Views and Beyond approach and illustrated the differences in formal precision.

We believe that by adopting a software language engineering approach for architectural viewpoints we have also shown the connection with software architecture design modeling and the fields of software language engineering and model-driven software development in general. We hope that this work has paved the way for further research in this direction.

In our future work we will apply the same approach to other architecture viewpoint frameworks. The V&B approach was a case study for us but we do not foresee serious obstacles in applying the same approach for other software architecture viewpoints and enterprise architecture viewpoints. We will elaborate on the tool and consider the integration of viewpoints for nonfunctional concerns. Further, we plan to enhance the tool for supporting architectural analysis.

Bibliography

[1] Acceleo – transforming models into code, http://www.eclipse.org/acceleo/, last accessed on Sep, 2012.

[2] Archimate 1.0 Specification, The Open Group, Tech. Rep. C091, Feb. 2009.

[3] J. Bézivin. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171-188, 2005.

[4] N. Boucké, D. Weyns, R. Hilliard, T. Holvoet, A. Helleboogh. Characterizing relations between architectural views. In: Proc. Of European Conference on Software Architecture (ECSA 2008). Paphos, Cyprus, September 29 - October 01, Springer, LNCS, 2008.

[5] A.W. Brown, Model driven architecture: principles and practice. SoSyM 3(3), 314–327, 2004.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. First Edition. Addison-Wesley, October 2002.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Second Edition. Addison-Wesley, 2010.

[8] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography, 2000.

[9] DoD Architecture Framework, U.S. Department of Defense, http://dodcio.defense.gov/dodaf20.aspx, last accessed on Sep, 2012.

[10] F. Durán and A. Vallecillo. Formalizing ODP Enterprise specifications in Maude. Computer Standards & Interfaces, vol. 25, no. 2, pp. 83–102, Jun. 2003.

[11] Eclipse Modeling Framework Web Site, http://www.eclipse.org/modeling/emf/, last accessed on Sep, 2012.

[12] Eclipse Modeling Framework Technology – EMFatic Project, http://www.eclipse.org/modeling/emft/?project=emfatic, last accessed on Sep, 2012.

[13] EuGENia, http://www.eclipse.org/gmt/epsilon/doc/eugenia/, last accessed Sep, 2012.

[14] M. Fowler, S. Scott, G. Booch. UML distilled, Object Oriented series, 179 p. Addison-Wesley, Reading, 1999.

[15] D. R. González, A. Vallecillo, J. R. Romero. On the Synchronization of ODP Textual and Graphical Specifications, In Proc. Of WODPEC 2010, pp. 376-381, Vitoria, Brazil, 2010.

[16] Eclipse Graphical Modeling Framework, http://www.eclipse.org/gmf/, last accessed on Sep, 2012.

[17] C. Hofmeister, R. Nord, and D. Soni. Applied Software Architecture. Addison-Wesley, NJ, USA.

[18] ISR, Institute for Software Reseach. ArchStudio 4.0 tool set for the xADL language, http://www.isr.uci.edu/projects/archstudio/, last accessed on Sep, 2012.

[19] [ISO/IEC 10746-2:1996] Information Technology - Open Distributed Processing - Reference Model: Foundations (ISO/IEC 10746-2). 1996.

[20] [ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010). (identical to ANSI/IEEE Std1471–2000), July 2007.

[21] J. Kienzle, N. Guelfi and S. Mustafiz. Crisis Management Systems A Case Study for Aspect-Oriented Modeling. In Transactions on aspect-oriented software development *VII*, Springer-Verlag, Berlin, 2010.

[22] F. Jouault, and I. Kurtev. Transforming Models with ATL, Model Transformations in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 Conference.

[23] A. Kleppe. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman Publishing Co., Inc., Boston, 2009.

[24] P. Kruchten. The 4+1 View Model of Architecture. IEEE Software, 12(6):42–50, 1995.

[25] T. Kuehne. Matters of (meta-) modeling. Software and System Modeling 5(4): 369-385, 2006.

[26] A.J. Lattanze. Architecting Software Intensive Systems: A Practitioner's Guide, Auerbach Publications, 2009.

[27] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages, IEEE Trans. Software Eng., vol. 26, no. 1, pp. 70–93, 2000.

[28] S.J. Mellor, K. Scott, A. Uhl, D. Weise. MDA Distilled: Principle of Model Driven Architecture, Addison Wesley, Reading , 2004.

[29] P.A. Muller, F. Fondement, and B. Baudry. Modeling Modeling, In: A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 2–16, 2009. Springer-Verlag Berlin Heidelberg, 2009.

[30] OMG: Model Driven Architecture (2003). http://www.omg.org/mda/, last accessed on Sep, 2012.

[31] OMG, MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10, 2002.

[32] C. Peña, J. Villalobos. An MDE Approach to Design Enterprise Architecture Viewpoints, IEEE 12th Conference on Commerce and Enterprise Computing (CEC), vol., no., pp.80-87, 2010.

[33] J. R. Romero, J. M. Troya, A. Vallecillo. Modeling ODP Computational Specifications Using UML, The Computer Journal 51: 435-450, 2008.

[34] N. Rozanski and E.Woods. Software Systems Architecture – Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2005.

[35] T. Stahl, M. Voelter. Model-Driven Software Development, Addison-Wesley, 2006.

[36] B. Tekinerdogan, C. Hofmann & M. Aksit. Modeling Traceability of Concerns for Synchronizing Architectural Views, in Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, pages 7–25, 2007.

[37] TOGAF 1995 -The Open Group Architecture Framework, Version 8.1.1., 1995.

[38] Xpand, Model-to-Text transformation language, http://www.eclipse.org/modeling/m2t/?project=xpand, last accessed on Sep, 2012.

[39] Xtext – Language Development Framework, http://www.eclipse.org/Xtext/, last accessed on Sep, 2011.

[40] J.A. Zachman. A Framework for Information Systems Architecture. IBM Systems Journal, Vol. 26. No 3, pp. 276-292, 1987.

Appendix A

Model-to-Text transformation template for decomposition viewpoint

```
«IMPORT org::xtext::example::mydsl::decomposition»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
__________________________________________________________________

  «name-» VIEW
__________________________________________________________________

  Stakeholders:

  Rationale:
__________________________________________________________________

Element Catalog
__________________________________________________________________

«FOREACH elements AS e SEPARATOR '\n'-»
«EXPAND writeElementInfo FOREACH elements-»
«ENDFOREACH»

«ENDFILE-»

«ENDDEFINE»

«DEFINE writeElementInfo FOR Element-»
«name-»:«FOREACH properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»

«EXPAND writeElementInfo FOREACH subelements-»
«ENDDEFINE»
```

Model-to-Text transformation template for uses viewpoint

```
«IMPORT org::xtext::example::mydsl::uses»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
```

```
______________________________________________________________________

  «name-» VIEW
______________________________________________________________________

  Stakeholders:

  Rationale:
______________________________________________________________________

Element Catalog
______________________________________________________________________

Elements:
«FOREACH elements AS e SEPARATOR '\n'-»
«e.name-»:«FOREACH e.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Relations:
«FOREACH uses AS u SEPARATOR '\n'-»
«u.source-» uses «u.target»«FOREACH pp.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for generalization viewpoint

```
«IMPORT org::xtext::example::mydsl::generalization»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
______________________________________________________________________

  «name-» VIEW
______________________________________________________________________

  Stakeholders:

  Rationale:
______________________________________________________________________

Element Catalog
______________________________________________________________________

Elements:
«FOREACH elements AS e SEPARATOR '\n'-»
«e.name-»:«FOREACH e.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
```

```
Relations:
«FOREACH relation AS r SEPARATOR '\n'-»
«r.source-» is a «r.target»«FOREACH pp.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for layered viewpoint

```
«IMPORT org::xtext::example::mydsl::layered»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
__________________________________________________________________

  «name-» VIEW
__________________________________________________________________

  Stakeholders:

  Rationale:
__________________________________________________________________

Element Catalog
__________________________________________________________________

Elements:
«FOREACH layers AS l SEPARATOR '\n'-»
«l.name-»:«FOREACH l.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Relations:
«FOREACH relation AS r SEPARATOR '\n'-»
«l.source-» allowed to use «l.target»«FOREACH pp.properties AS p-
»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for aspects viewpoint

```
«IMPORT org::xtext::example::mydsl::aspects»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
```

```
________________________________________________________________________

  «name-» VIEW
________________________________________________________________________

  Stakeholders:

  Rationale:
________________________________________________________________________

Element Catalog
________________________________________________________________________

Elements:
«FOREACH elements AS e SEPARATOR '\n'-»
«e.name-»:«FOREACH e.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Relations:
«FOREACH crosscuts AS c SEPARATOR '\n'-»
«c.source-» crosscuts «c.target» : «c.detail.expl»
«FOREACH pp.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for pipe and filter viewpoint

```
«IMPORT org::xtext::example::mydsl::pipe_and_Filter»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
________________________________________________________________________

  «name-» VIEW
________________________________________________________________________

  Stakeholders:

  Rationale:
________________________________________________________________________

Element Catalog
________________________________________________________________________

Filters:
«FOREACH filters AS f SEPARATOR '\n'-»
«f.name-»:«FOREACH f.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
```

```
Pipes:
«FOREACH pipes AS pp SEPARATOR '\n'-»
«pp.name-»:Flows data from «pp.filter1.name-» to «pp.filter2.name-
»«FOREACH pp.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for shared data viewpoint

```
«IMPORT org::xtext::example::mydsl::shared_data»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
__________________________________________________________________

  «name-» VIEW
__________________________________________________________________

  Stakeholders:

  Rationale:
__________________________________________________________________

Element Catalog
__________________________________________________________________

Repositories:
«FOREACH repository AS r SEPARATOR '\n'-»
«r.name-»:«FOREACH r.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
DataAccessors:
«FOREACH dataAccessor AS da SEPARATOR '\n'-»
«da.name-»:«FOREACH da.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Attachments:
«FOREACH dataRead AS dr SEPARATOR '\n'-»
«dr.name-»:Reads data from «dr.rp.name-» via «dr.da.name-»
«FOREACH pp.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«FOREACH dataWrite AS dw SEPARATOR '\n'-»
«dw.name-»:Writes data to «dw.rp.name-» via «dw.da.name-»
«FOREACH pp.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
```

```
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for client server viewpoint

```
«IMPORT org::xtext::example::mydsl::client_server»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
____________________________________________________________________

  «name-» VIEW
____________________________________________________________________

  Stakeholders:

  Rationale:
____________________________________________________________________

Element Catalog
____________________________________________________________________

Servers:
«FOREACH server AS s SEPARATOR '\n'-»
«s.name-»:«FOREACH s.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Clients:
«FOREACH client AS c SEPARATOR '\n'-»
«c.name-»:«FOREACH c.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

Attachments:
«FOREACH attachment AS at SEPARATOR '\n'-»
«at.name-»:Attaches «at.client.port-» to «at.server.port-»
«FOREACH pp.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for deployment viewpoint

```
«IMPORT org::xtext::example::mydsl::deployment»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
```

```
«FILE "archdoc.doc"-»
______________________________________________________________________

  «name-» VIEW
______________________________________________________________________

  Stakeholders:

  Rationale:
______________________________________________________________________

Element Catalog
______________________________________________________________________

Elements:

«FOREACH hardwareElements AS ee SEPARATOR '\n'-»
«ee.name-»:«FOREACH se.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«EXPAND writeElementInfo FOREACH softwareElements-»
«ENDFOREACH»
«ENDFOREACH»

«DEFINE writeElementInfo FOR SoftwareElement-»
«FOREACH softwareElements AS se SEPARATOR '\n'-»
«se.name-»:«FOREACH se.properties AS p-»«p.value-»
   «IF p.name == "expl"-»
   «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDDEFINE»

«ENDFILE-»
«ENDDEFINE»
```

Model-to-Text transformation template for install viewpoint

```
«IMPORT org::xtext::example::mydsl::install»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
______________________________________________________________________

  «name-» VIEW
______________________________________________________________________

  Stakeholders:

  Rationale:
______________________________________________________________________

Element Catalog
______________________________________________________________________

«FOREACH directories AS d SEPARATOR '\n'-»
«EXPAND writeElementInfo FOREACH directories-»
```

```
«ENDFOREACH»

«ENDFILE-»

«ENDDEFINE»

«DEFINE writeElementInfo FOR Directory-»
«name-»:«FOREACH properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»

«EXPAND writeElementInfo FOREACH files-»
«EXPAND writeElementInfo FOREACH components-»

«ENDDEFINE»
```

Model-to-Text transformation template for work assignment viewpoint

```
«IMPORT org::xtext::example::mydsl::work_assignment»
«EXTENSION templates::Extensions»

«DEFINE main FOR Model-»
«FILE "archdoc.doc"-»
_________________________________________________________________

  «name-» VIEW
_________________________________________________________________

  Stakeholders:

  Rationale:
_________________________________________________________________

Element Catalog
_________________________________________________________________

Elements:
«FOREACH softwareElements AS se SEPARATOR '\n'-»
«se.name-»:«FOREACH se.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»

«FOREACH environmentalElements AS ee SEPARATOR '\n'-»
«ee.name-»:«FOREACH se.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»


Allocations:
«FOREACH allocation AS a SEPARATOR '\n'-»
«a.software-» is allocated to «a.environment»«FOREACH
pp.properties AS p-»«p.value-»
  «IF p.name == "expl"-»
  «ENDIF-»
«ENDFOREACH»
«ENDFOREACH»
«ENDFILE-»
«ENDDEFINE»
```

Publications Related to This Thesis

1. E. Demirli, B. Tekinerdogan. Software Language Engineering of Architectural Viewpoints, in Proc. of the 5th European Conference on Software Architecture (ECSA 2011), LNCS 6903, pp. 336–343, 2011.

2. E. Demirli, B. Tekinerdogan. SAVE: Software Architecture Environment for Modeling Views. In Proc. of the 2011 9th Working IEEE/IFIP Conference on Software Architecture(WICSA '11). IEEE Computer Society, Washington, DC, USA, 355-358., 2011.

3. B. Tekinerdogan, E. Demirli. Evaluation Framework for Software Architecture Viewpoint Approaches. Software and Systems Modeling, to be submitted.