A Model Driven Approach for Automating Architecture Documentation Process

Başak Çakar, Elif Demirli and Şadiye Kaptanoğlu
Department of Computer Engineering, Bilkent University
06800 Bilkent, Ankara, Turkey
{cakar, demirli, sadiye}@cs.bilkent.edu.tr

Abstract

Software architecture documentation is a heavyweight process since it involves development and management of various documentation artifacts. Architectural views and architecture description languages (ADLs) are two examples of those artifacts each of which presents mostly similar information in different forms. So, there is a redundant documentation effort when two models are developed separately. In this paper, we present a model-driven approach for automatically transforming architectural views to AADL models. Architectural views that are defined based on our proposed metamodel is transformed to AADL model. In addition to elimination of redundant documentation effort, this approach also enhances architecture documentation process by ensuring consistency among views and AADL models.

1. Introduction

Software architecture for a 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 [1]. Documentation of software architecture plays an important role since it facilitates early communication of design decisions and system analysis. A complete architecture description process is involves various documentation activities.

Architectural view is one such type of documentation. Since modern systems are too complex to document and communicate with a single architecture model, architectural view concept was introduced. An architectural view is a description of one aspect of a system's architecture. It represents a set of system elements and relations associated with them to support a particular concern [1]. Having multiple views helps to separate the concerns and support the modeling, communication and analysis of the software architecture with different stakeholders.

In the literature, various multi-view approaches are defined to document the architecture. For example, Rational's Unified Process is based on Kruchten's 4+1 view approach which comprises logical, development, process view and physical views [2]. The Siemens Four Views model [3] is another example.

Current trend in architectural views is to enable architects to produce whatever views are useful for the system at hand. The Views and Beyond (V&B) is in alignment with this trend. It is a multi-view approach that views the system from three different viewtypes: Module viewtype, component-and-connector (C&C) viewtype and allocation viewtype.

Documenting the system from different views is useful; however, a complete and unified description of the architecture is also required. Architecture description languages (ADLs) are used for this purpose. They facilitate design, analysis and documentation for the overall system. There are various architecture description languages in the literature. Each one is developed for different purposes.

AADL (Architecture Analysis and Design Language) is a popular architecture description language [4]. It provides formal modeling concepts for the description and analysis of application systems. The AADL includes software, hardware, and system component abstractions to specify and analyze real-time embedded systems. It allows mapping software onto computational hardware elements [4].

Both ADL and architectural views are crucial in architecture documentation process. They mostly capture the same information; however, the presentation of information is different. So, a redundant work is done when two artifacts are developed separately. Also, a considerable amount of effort is required to keep the two documents consistent.

In this paper, we present a model-driven approach for automating ADL document generation from architectural views. The metamodel for architectural views is developed for V&B approach. Then, model-to-model transformation is applied using the existing AADL metamodel. A model-to-text transformation is also applied to present a textual specification of architectural views.

The remainder of the paper is organized as follows: Problem statement is presented in Section 2. Section 3 gives Domain Analysis for developing V&B metamodel. Section 4 gives the grammar for the designed domain specific language. The metamodel for V&B approach is defined from scratch based on MOF in Section 5. It presents abstract syntax, concrete syntax, and static semantics for the domain and gives an example model. In Section 6, the metamodel is developed using UML profiling. Section 9 presents lessons learned and conclusions.

2. Problem Statement

Documenting software architectures is a heavyweight task since it involves various activities. For example, the architecture should be documented for different views. However, a complete and unified description of architecture is also required.

In both view documents and architecture description the same information is presented in different forms. When they are done separately, there is some redundant effort. In addition to this, it must be ensured that the information presented in different documentation artifacts is consistent. An additional effort to keep the artifacts consistent is also required. In order to reduce architecture documentation effort, automatic transformation techniques should be applied.

3.1. Architectural Viewtypes and Styles

In order to document software architectures, three kinds of viewtypes are defined in V&B approach. A viewtype defines the element and relation types that are used to describe the architecture of a software system from a particular perspective [1]. In V&B approach domain, there are three basic viewtypes: module viewtype, component and connector viewtype, and allocation viewtype.

Each of these viewtypes considers the architecture from a different point of view. Module viewtype is about how the system is structured as a set of implementation units. So, a module is an implementation unit of software that provides a coherent unit of functionality [1]. Considering the C&C viewtype, components and connectors are used to describe the run-time behavior and interactions of a software system. The allocation viewtype is used to express the allocation of software elements to its development and execution environments.

In addition to these three viewtypes, within each viewtype there are commonly occurring forms and variations. These are called architectural styles or simply styles. A style is defined as a specialization of elements and relationships, together with a set of constraints on how they can be used [1]. There are a number of predefined styles for each of the three viewtypes defined above. For example, the styles of the module view are: decomposition style uses style, generalization style, and layers style. However, one can also define a new style to satisfy the needs of software project development and documentation.

3.2. Domain Concepts

Software architecture: software architecture of a system is the structure or structures of the system which comprise software components, or the documentation of these.
Viewtype: a viewtype is the definition of element and relation types that are used to describe the architecture of a software system from a particular perspective. There are three viewtypes according to V&B approach: Module, C&C and Allocation.
Style: a style is a specialization of element types and relation types along with any constraints. Each style conforms to one viewtype. Any number of architectural styles can be defined.
View: a view is a representation of the elements of a system and their relations.
Element: an element of a system is the type of one of the organizational units of the system used in the documentation.
Relation: a relation is a pattern of interaction among two or more elements.
Property: a property is an attribute of either an element or a relation.

4. DSL Grammar

Domain specific language is used to represent concepts and rules of a particular domain. In order to describe domain concepts in a formal way BNF can be used. Backus–Naur Form (BNF) is a formal notation used to describe the syntax of a given language.

Figure 1 shows the mapping of the domain concepts of Views and Beyond Approach to a domain specific grammar in EBNF (Extended Backus–Naur Forms) that is the extension of BNF. This EBNF grammar can be interpreted as follows: our root element is Architecture which has zero or more ViewType. ViewType has a ViewTypeName and zero or more Style and so on. In EBNF grammar non-terminals consist of non-terminals and terminals.

*Architecture ::= (Viewtype)**
*Viewtype ::= ViewtypeName (Style)**
ViewtypeName ::= VTName
*Style ::= StyleName Topology (ArchitecturalElement)**
StyleName ::= String
Topology ::= String
ArchitecturalElement::= ArchitecturalElementName (Property) (Element | Relation)*
ArchitecturalElementName ::= String
Property ::= PropertyName PropertyValue
PropertyName ::= String
PropertyValue ::= String
*Element ::= ElementName (Relation)**
ElementName ::= String
Relation ::= TargetElement
TargetElement ::= String

Figure 1. Domain specific grammar of V&B Approach.

5. Metamodel Based on MOF

One way to define a metamodel is using Meta Object Facility (MOF) that is a language used to define metamodels. Metamodel describes concepts that can be used for modeling the model. A complete metamodel consists of abstract syntax of the domain, concrete syntax and static semantics. In this section we present the metamodel of architectural views based on MOF-from scratch. We used Eclipse Modelling Framework (EMF) [5] TOPCASED [6] plug-in in order to construct our metamodel. In the following subsections we will explain the abstract syntax, concrete syntax, static semantics and example models in detail.

5.1. Abstract Syntax

Abstract syntax consists of the concepts and definition of a domain specific language. It represents domain concepts and the relationship between these concepts.

In Figure 2, abstract syntax of architectural views is shown. Main entity of the metamodel is Architecture. It consists of zero or more Viewtype. Viewtype has viewtype_name as an attribute which can be Module, C&C or Allocation. Collection of Style conforms to Viewtype. Attributes of Style are styleName and topology that is used defines the constraints about related Style. Each Style consists of zero or more ArchitecturalElement. Each architectural element has zero or more Property. ArchitecturalElement in the style can be Element or Relation. According to this model each Relation must be between two elements which are source and target.

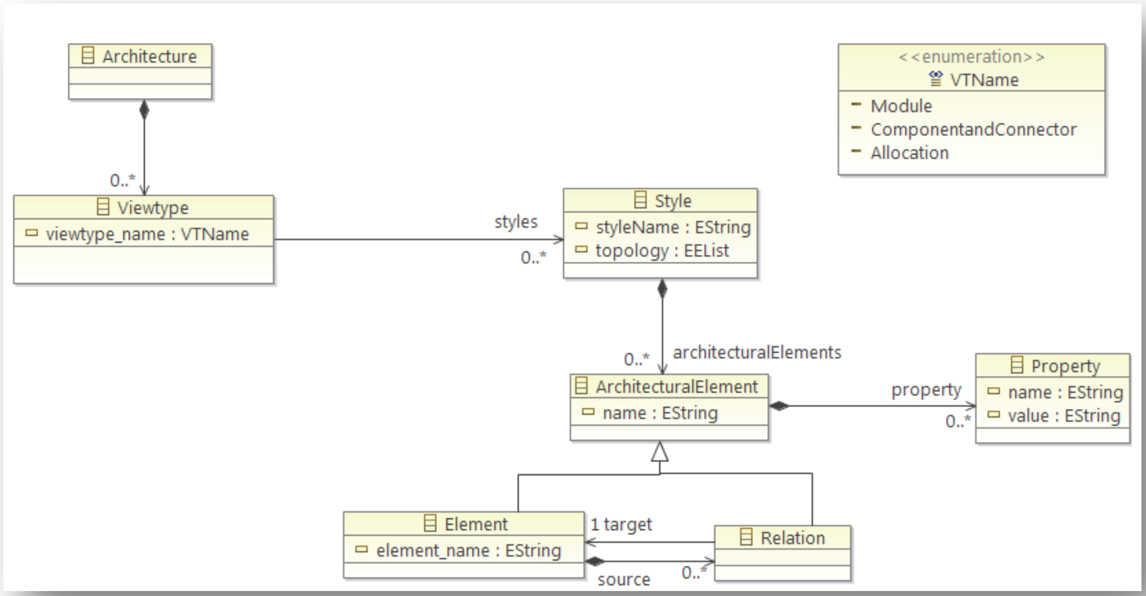


Figure 2. Metamodel based on MOF from scratch

5.2. Concrete Syntax

Concrete Syntax is realization of abstract syntax and it can be visual or textual. The visual concrete syntax of our metamodel is expressed in Figures 3 and 4. Visual concrete syntax shows how to represent domain concepts which are in abstract syntax. Both standard architectural views such as Module, C&C and Allocation and new viewtypes can be built by using our metamodel. Thus different concrete syntax can be defined for each of them separately. We defined our concrete syntax based on the three viewtypes of V&B approach. We tried to use the common representation of styles in these viewtypes which consists of basic UML notations [1].

5.2.1. Module Viewtype

Figure 3 shows how to represent Element and Property in module viewtype. In module view an element can be a class, a package, a layer or any decomposition of the code unit. ElementName and ElementType enable to give name to element and to define its type. In this figure we see that the given element is a Module and its name is Database. By using the property representation properties and their values of related element are shown. In the example property name is “VisibleTo” which shows Database Module is visible to Implementation Module.

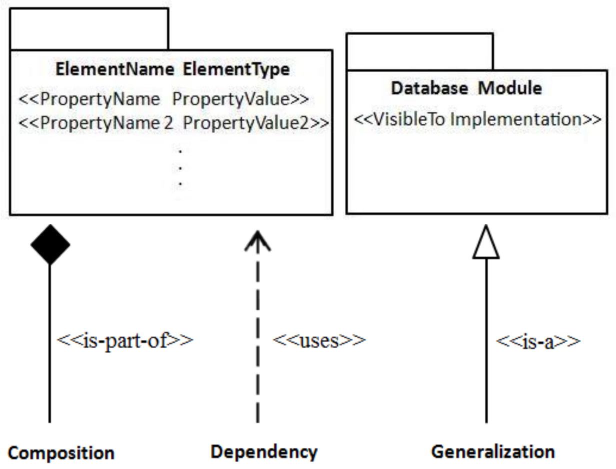


Figure 3. Representation of element and relation in module viewtype

Figure 3 also shows the representation of relations in module view. In module view there are three types of relations which are composition, dependency and generalization between elements.

5.2.2. C&C Viewtype

C&C viewtype provides to represent software architecture from the perspective of its components runtime interactions of principle units and its connectors [7]. Figure 4 shows how to represent C&C viewtype’s elements and relations. In C&C viewtype there are two types of element, Component and Connector. Components are the principle processing unit of the executing system and connectors shows the interaction mechanism between components. Each element has ports which provide interaction of components and connectors through their interfaces and define a set of operations and events that are provided by the element and that are required from its environment. There are two types of interfaces, Required Interface and Provided Interface. A provided interface is modeled using the lollipop notation and a required interface is modeled using the socket notation.

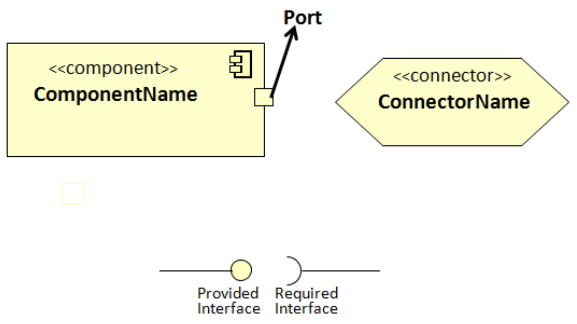


Figure 4. Representation of Element and Relation in C&C Viewtype

5.2.3. Allocation Viewtype

The allocation viewtype is used to show mapping of the software architecture onto its environment which can be hardware elements, file management or organization team. Concrete syntax of allocation viewtype differs too much compared to other viewtypes. For each style such as Deployment Style, Implementation Style and Work Assignment Style we can define different concrete syntax. For simplicity and understandability we prefer to define our concrete syntax based on deployment style. In Deployment Style software element that are elements from C&C viewtype and environmental element (computing hardware such as processor, memory, disk, etc.) are used. “Allocated-to” and “Migrates-to” are used as relation. Allocated-to shows on which physical elements the software resides. Migrates-to shows the relation from a software element on one processor to the same software element on a different processor, this relation indicates that a software element can move from processor to another and it is used when the allocation is dynamic [1]. Figure 5 shows how to represent elements and relations of Deployment Style. In the figure concrete syntax of software element is the same with the concrete syntax of C&C viewtype because Deployment Style uses C&C elements as software element.

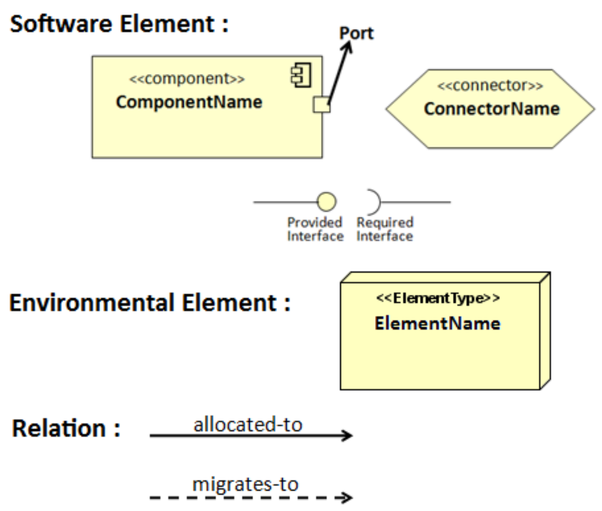
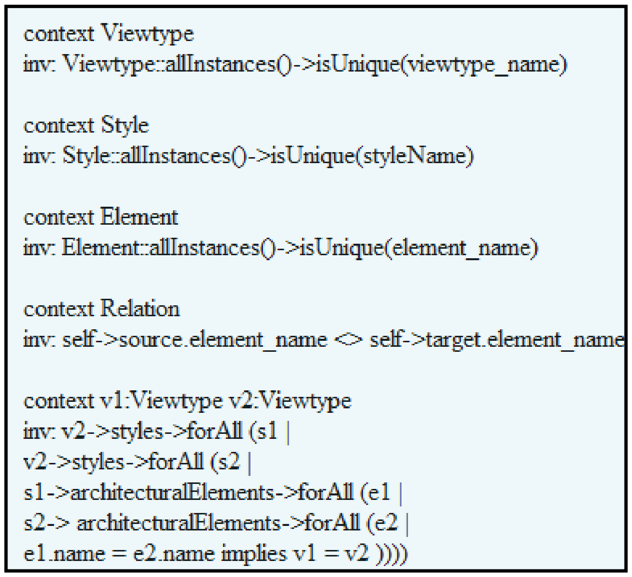


Figure 5. Representation of Element and Relation in Allocation Viewtype

5.3. Static Semantics

The static semantics of the metamodel shown in Figure 6 are presented in this section. In order to define the static semantics we used object constraint language (OCL). We have a limited list of well-formedness rules as seen in Figure 6. The first three rules define the uniqueness property of names. Viewtype name, style name and element names should be unique in our domain. Next rule states that a relation should always occur between two different elements. An element cannot have a relation to itself. The last rule indicates that styles cannot be mixed for a specific viewtype. To be more specific, one cannot use the architectural elements defined in a style of one viewtype in a style of another viewtype.



```
context Viewtype
inv: Viewtype::allInstances()->isUnique(viewtype_name)

context Style
inv: Style::allInstances()->isUnique(styleName)

context Element
inv: Element::allInstances()->isUnique(element_name)

context Relation
inv: self->source.element_name <> self->target.element_name

context v1:Viewtype v2:Viewtype
inv: v2->styles->forAll (s1 |
v2->styles->forAll (s2 |
s1->architecturalElements->forAll (e1 |
s2-> architecturalElements->forAll (e2 |
e1.name = e2.name implies v1 = v2 ))))
```

Figure 6. OCL rules for metamodel from viewtype

5.4. Example Model

Figure 7 shows the application of uses style on the Crime Management System (CMS). CMS is an online system that aims to enable collaboration among police and citizens while fighting with crime. It is used by citizens to make denouncements to the police and used by police officers to properly report and manage crime, to make crime analysis etc. The architecture of the system is modeled with uses style in order to see which modules use the others and thus make an incremental development plan of the system.

The mapping between the defined abstract syntax and the example model can be investigated on a smaller example in Figure 8. The basic architectural element used in this representation is module and it expressed by UML package diagram notation. The “Module” keyword written inside each unit emphasized its type. It is defined by the name attribute of the ArchitecturalElement abstraction in Figure 7. The string before “Module” keyword is name of the module. It is represented as the attribute of Element class in the abstract syntax. In the example, we have two modules: Statistics and Lost Citizen. The relation

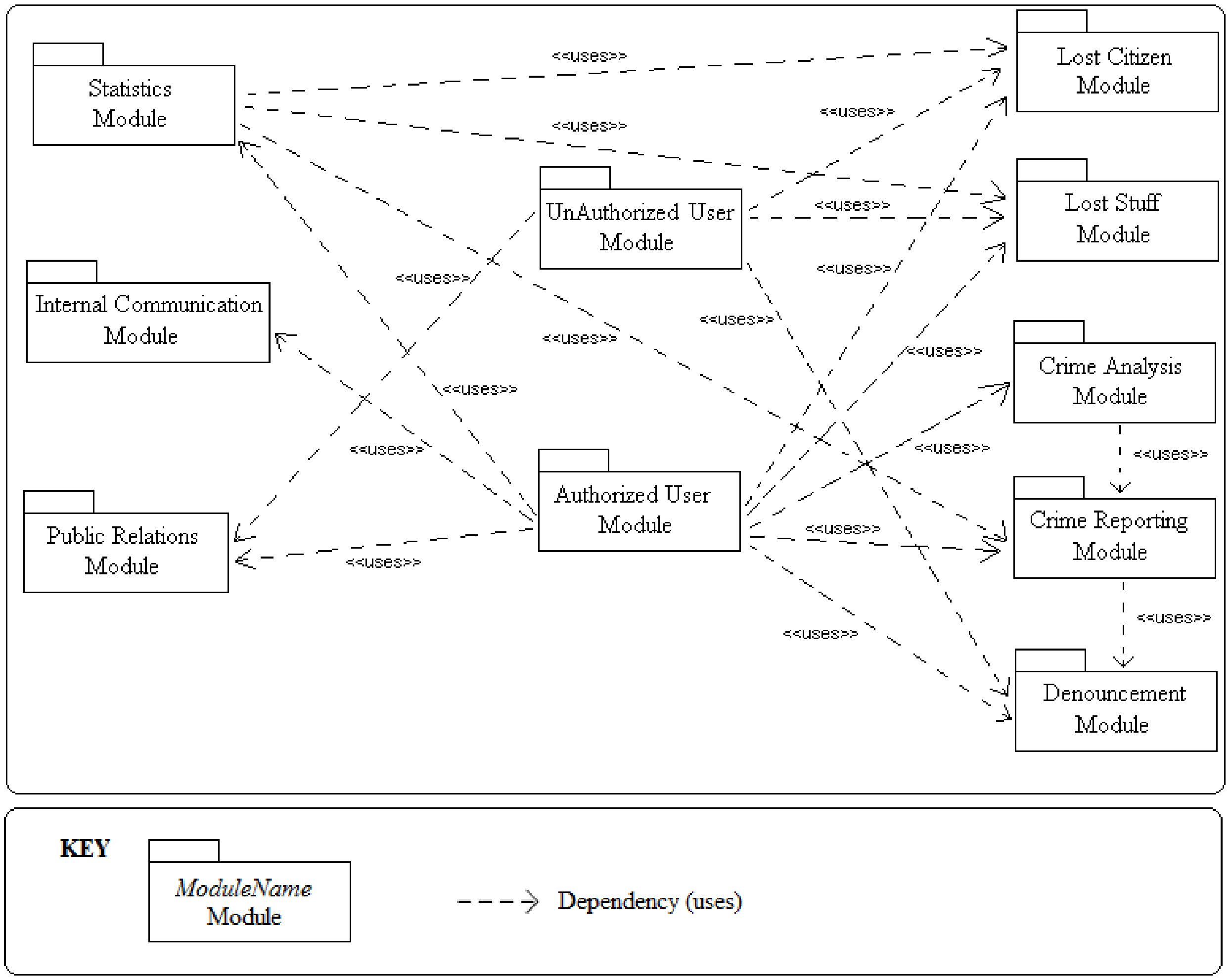


Figure 7. A sample model for metamodel from scratch – Uses Style for Crime Management System

between those modules is a uses relation. "uses" keyword is expressed by name attribute of ArchitecturalElement abstraction similarly to "module".

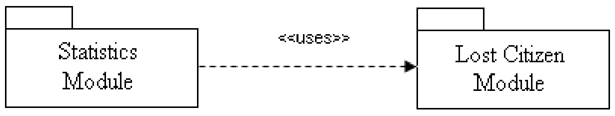


Figure 8. A simple example to uses relation

6. Metamodel Using UML Profiling

The second approach of defining a metamodel is using an existing available metamodel and extending its definition. In this section, we present the metamodel for architectural views by extending the UML metamodel. Although it is possible to make a heavyweight extension, we preferred to develop a lightweight extension of UML metamodel since the resulting metamodel of a heavyweight extension is not recognized by existing UML tools. In the following subsections, the abstract syntax, concrete syntax, static semantics and two example models are presented.

6.1. Abstract Syntax

The abstract syntax of our metamodel based on UML profiling is seen in Figure 9. We performed a UML 2.* lightweight extension. The basic domain concepts identified in domain analysis are mapped to UML concepts in this metamodel. By this way, we extend existing UML concepts to satisfy the domain concept requirements.

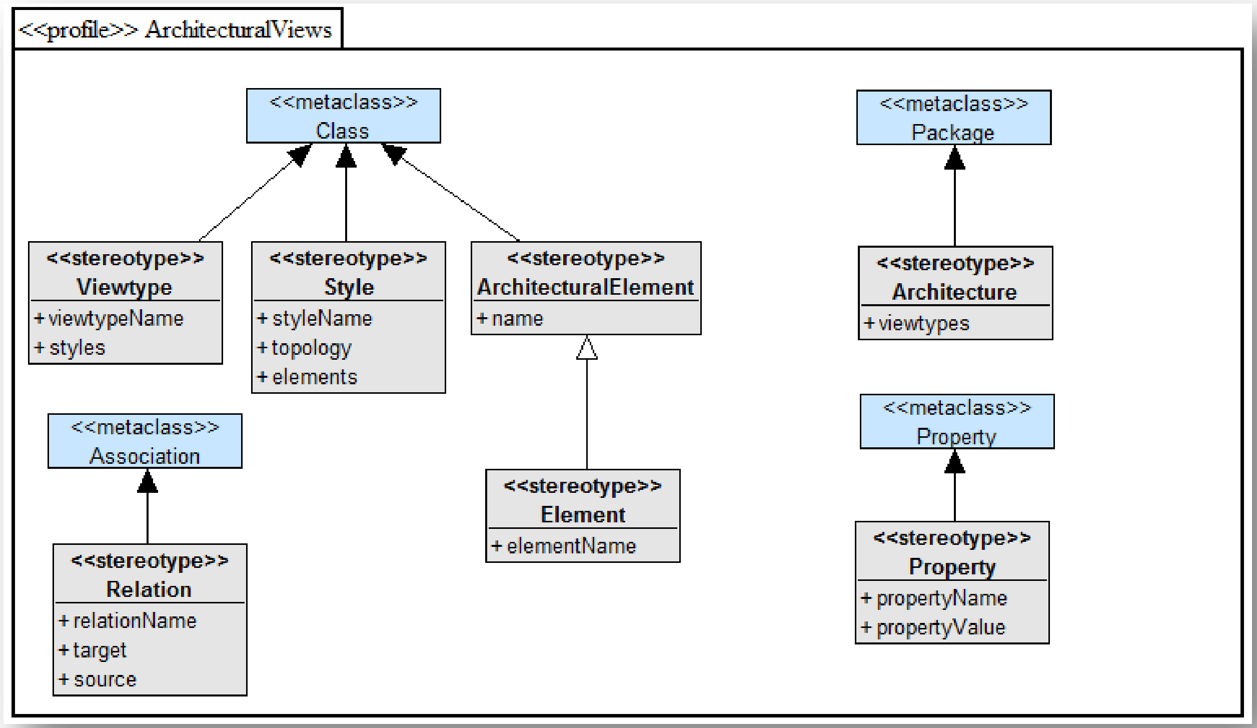


Figure 9. Metamodel based on UML profiling

The architecture concept is extended from UML Package since it acts as a covering mechanism rather than being a separate class with its own properties. The viewtypes that constitute the architecture are extended from UML Class. Moreover, the styles that reside in the viewtype are also mapped to UML Class. Thus, each style will have its own class and attributes. The basic building blocks of styles are called architectural elements and they are also mapped to UML class. In our previous metamodel which was based on MOF from scratch, architectural element was the generalization of element and relation. In this metamodel, architectural element stereotype is again a generalization of element stereotype. However, relation concept is mapped to UML Association which is a more suitable UML concept to its behavior. Lastly, property concept is extended from UML Property metaclass which adds propertyName and propertyValue attributes.

6.2. Concrete Syntax

In the previous section, we have used the extension of an available metamodel method in order to create architectural views metamodel. This extension mechanism offers five kinds of concrete syntax definitions based on UML's concrete syntax [8]. These are simply: 1) showing as direct instance of metaclass, 2) using the name of metaclass as a stereotype, 3) using an abbreviation by convention as stereotype, 4) using a tagged value stating the metaclass, 5) creating an individual graphical notation. Among these five options, we see that option number three is the most practical and used this option to create our concrete syntax.

The concrete syntax for the metamodel created using UML profiling is seen in Figure 10. In the figure it is seen that a graphical notation very similar to UML is used. For concepts extended from UML class we use class notation together with a stereotype of abbreviation by convention. In the figure, it is shown for Element only. For the architecture concept which is extended from UML Package, the same approach is used. However, the property is shown as in the UML concrete syntax. Lastly, relation concept is almost same with the UML relation. Since we extended the UML association, the relation types are same. In addition, relation names should be written on the relations together with multiplicities where necessary.

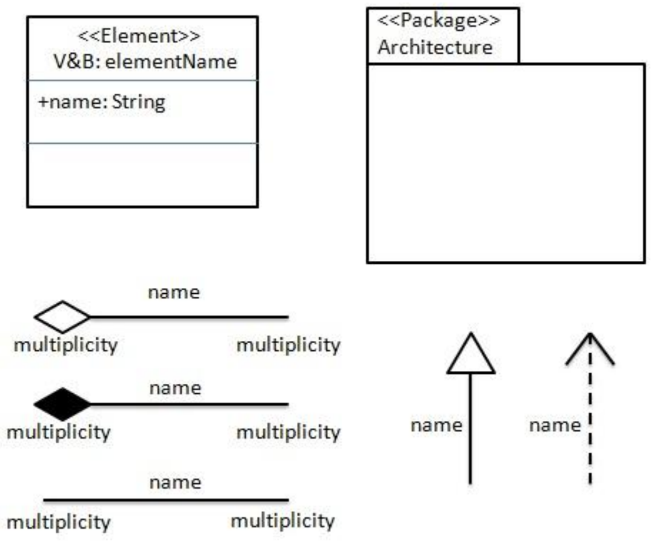


Figure 10. Concrete syntax for UML profiling based metamodel.

6.3. Static Semantics

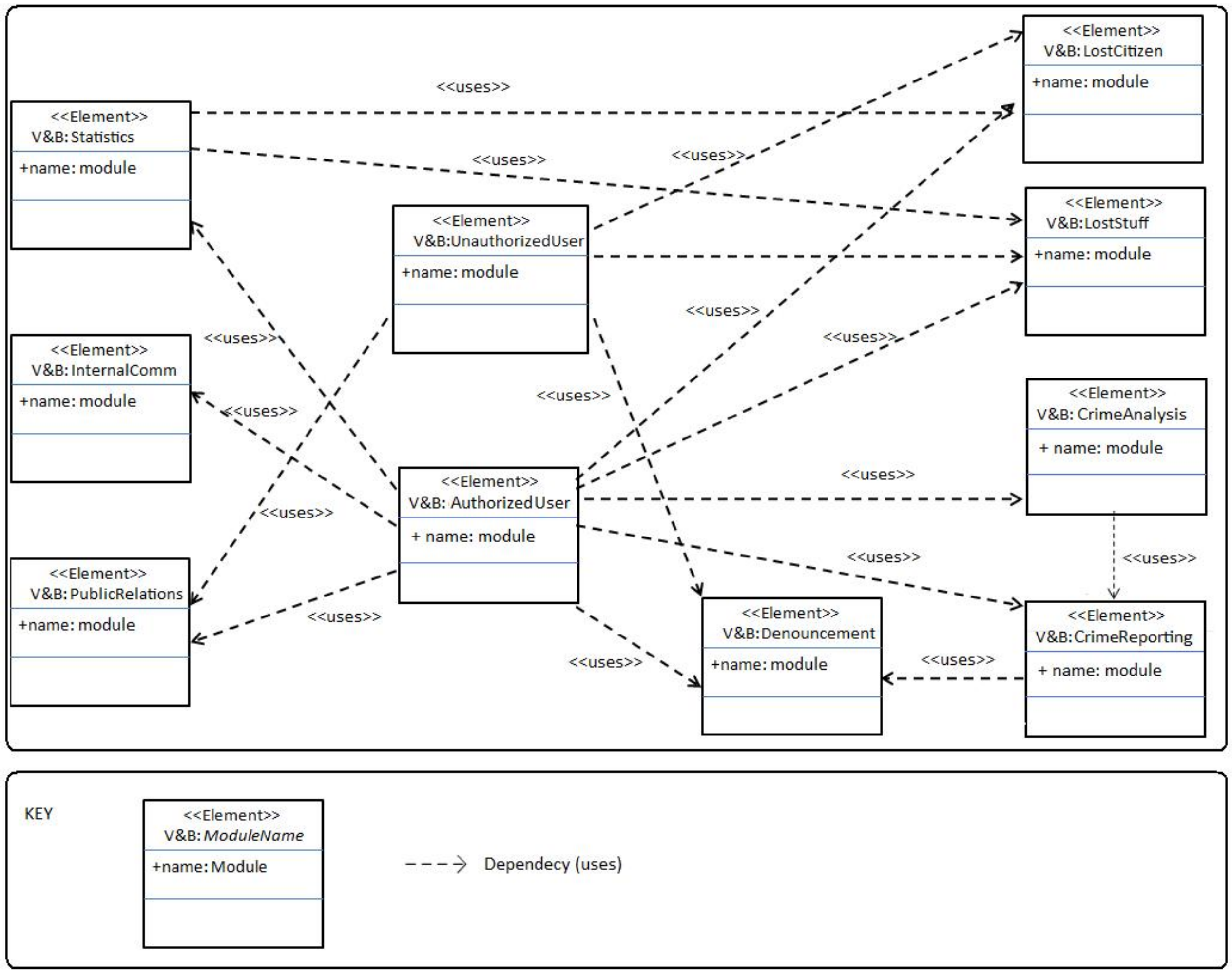
The static semantics of the metamodel is presented in this section. In order to define static semantics with UML profiling we use notes in the metamodel description. The constraints are written in OCL and can be seen below. As it is seen, most of the rules are same with the previous rules which were defined for metamodel from scratch. This is because we are at the same domain. We needed to add additional rules because with UML profiling the expressiveness of the metamodel decreases. We add additional rules for describing the relations between domain concepts. For example, a style can belong to one viewtype only. In addition, since the types of properties are not declared for stereotypes, we defined rules for these. For example, the target and source for a relation is of Element type.

Static Semantics of Metamodel with UML Profiling

```
context Viewtype
inv: Viewtype::allInstances()-
>isUnique(viewtypeName)

context Style
inv: Style::allInstances()->isUnique(styleName)

context Element
inv: Element::allInstances()->isUnique(elementName)

context Relation
inv: self.source.elementName <>
self.target.elementName

context v1:Viewtype v2:Viewtype
inv: v1.styles->forAll (s1 |
        v2.styles->forAll (s2 |
        s1->elements->forAll (e1 |
        s2-> elements->forAll (e2 |
        e1.name = e2.name implies v1 = v2 ))))

context Viewtype
inv: self.viewtypeName = 'Module' or
        self.viewtypeName =
'ComponentAndConnector' or
        self.viewtypeName = 'Allocation'

context  Architecture
inv: self.viewtypes->size() = 3

context v1: Viewtype v2:Viewtype
inv: v1.styles->forAll (s1 |
        v2.styles->forAll (s2 |
        s1.styleName = s2.styleName implies v1=v2))

context Relation
inv: self.target->oclIsTypeOf (Element) and
        self.source-> oclIsTypeOf (Element)

context Architecture
inv: self.viewtypes->forAll ( v1 | v1->oclIsTypeOf
(Viewtype))

context Viewtype
inv: self.styles->forAll (s1 | s1->oclIsTypeOf (Style))

context Style
inv: self.aelements->forAll (e1 |
        e1->oclIsTypeOf (ArchitecturalElement))
```

6.4. Example Model

In Figure 11, the uses view that is shown in Figure 7 is re-modeled using the newly concrete syntax that was created with UML profiling. The <<Element>> stereotype shows that that unit is an element. In M2, it is defined extending the UML Class object. It has a name and also a name attribute. The second name here implies the type of the unit such as module, package etc. The dependency relations here are extended from UML Association class. It has a name to describe the relation which is "uses" in Figure 11.



<<Element>> V&B: Statistics +name: module
<<Element>> V&B: InternalComm +name: module
<<Element>> V&B: PublicRelations +name: module
<<Element>> V&B:UnauthorizedUser +name: module
<<Element>> V&B: AuthorizedUser + name: module
<<Element>> V&B: LostCitizen +name: module
<<Element>> V&B: LostStuff +name: module
<<Element>> V&B: CrimeAnalysis + name: module
<<Element>> V&B:CrimeReporting + name: module
<<Element>> V&B:Denouncement +name: module
<<uses>>
KEY
<<Element>> V&B: ModuleName +name: Module
Dependecy (uses)

Figure 11. A sample model for metamodel based on UML profiling – Uses Style for Crime Management System

7. Model-to-Model Transformation

A model transformation takes as input a model conforming to a given metamodel and produces as output another model conforming to another given metamodel. In this work, we defined rules for transforming our models to AADL models. Thus, the role of our model-to-model transformation is basically mapping among models at different levels of abstraction. We applied exogenous transformation meaning that the two metamodels used in the transformation are expressed in different languages.

The AADL is a standard which provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions. It includes software, hardware, and system component abstractions to mainly specify and analyze real-time embedded systems, complex systems of systems, and specialized performance capability systems, and map software onto computational hardware elements [4].

In the following two sub-sections, we present the definition of our model-to-model transformation and an example transformation.

7.1. Definition

In order to define the model-to-model transformation, we used our MOF from-scratch metamodel, as source and the AADL (The Architecture Analysis and Design Language) metamodel as target. In our metamodel there are concepts such as architecture, style, element, and relation which are nice abstractions. Whereas, the AADL metamodel consists of seven main concepts each of which are defined in a separate metamodel definition file. The combination of these sub-metamodels forms the metamodel which is a highly complex structure.

For this work, we defined model-to-model transformation rules for the Component-and-Connector viewtype which has more similar concepts to AADL's metamodel. We considered the predefined styles of this viewtype which are listed in [1] as: pipe-filter style, client-server style, peer-to-peer style, and etc. We defined mappings and transformation rules considering each of these styles and concepts defined in them. The concept of these mappings can be seen in Table 1.

Table 1. Mapping of metamodels for transformation

Concepts (our metamodel)	**Concepts (AADL metamodel)**
Architecture	AadlSpec
Style	AadlPackage
Element	Component (ProcessType, DataType, busType)
Relation	Connection
Property	Feature

For example Architecture concept is mapped to AadlSpec in AADL and its name is also transformed to the name of AadlSpec name. The table does not show the details of transformations, but it is for a general idea of mapping. We defined rules considering the different styles of Component-and-Connector viewtype. For instance, Element is transformed to Component with considerations on its name. If the element name is "Pipe" or "Filter" the Component is of processType, or if the element name is publish-subscribe then it is transformed to Component of busType.

7.2. Example

In order to perform model-to-model transformation, we created the model of "MergeAndSort" according to our from-scratch metamodel. This model can be seen in Figure 12. In this model, there are four elements of type filter and four elements of type pipe. All these elements have features which define the port used for connection and communication. The source model conforms to PipeAndFilter style of Component and Connector viewtype. When we apply the developed transformation to this source model, we get the target model in Figure 14 which conforms to AADL metamodel. This model is developed by using the Eclipse plugin for AADL development. We give the target model of transformation and this plugin generates the concrete syntax representation of our model. This is also a verification step since it generates concrete syntax of valid input models only. From this figure, we can see that the elements are mapped to components according to their names. In this case both pipes and filters are mapped to ProcessType components. The ports of each component are also shown in the concrete syntax.

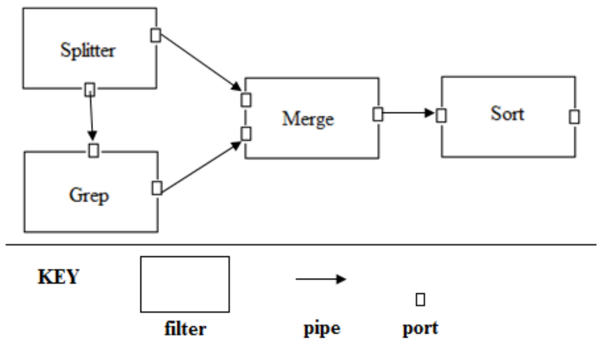


Figure 12. Merge-and-Sort model in C&C view

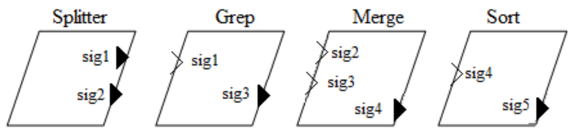


Figure 13. Merge-and-Sort model with AADL

8. Model-to-Text Transformation

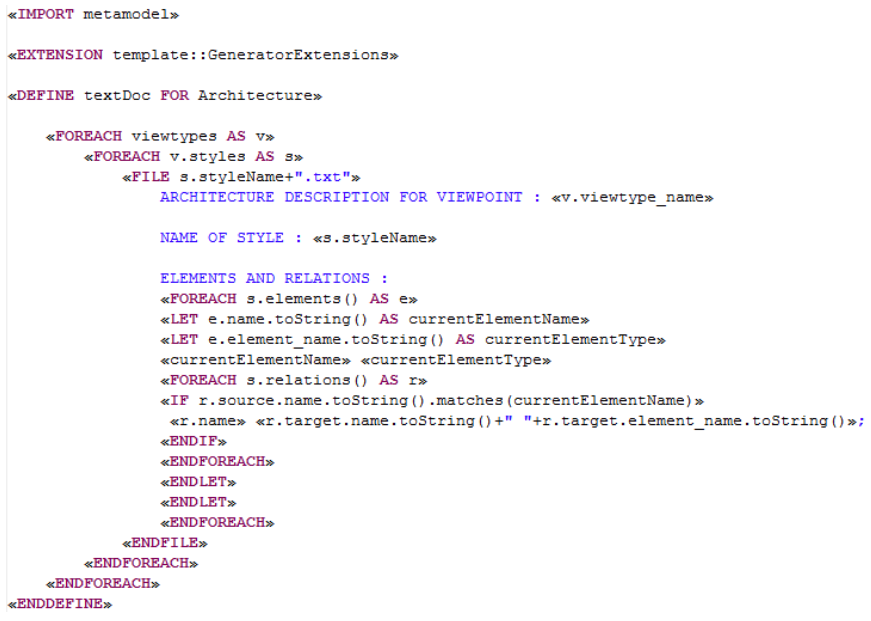
8.1. Definition

Model-to-text transformation is a 'special' case of model-to-model transformation. It provides developers to generate both code and non-code documents such as documents and plays an important role in the Model-Driven Software Development (MDSD). In our project we prefer to transform our model into a text document since in software development documenting textual specifications is required for stakeholders and development team. Thus generating textual representation of architectural models automatically provides more consistent and easy documentation.

In this study we used openArchitectureWare (oAW) as transformation engine. oAW consist of M2M transformations, constraints checking, a workflow engine, adapters for the XMI of a variety of UML tools, EMF integration, Eclipse IDE integration as well as a proven template language for code generation called Xpand. oAW requires a metamodel and a model which is formed based on that metamodel for transformation. It first validates the model according to predefined check rules that are defined by an OCL–like language. A template file is written by Xpand in order to define the format of output file by mapping model elements to text segments and oAW generates the target document by using this file. All of these are stored in a workflow file that directs oAW engine by instructing which tasks will be executed under which configurations.

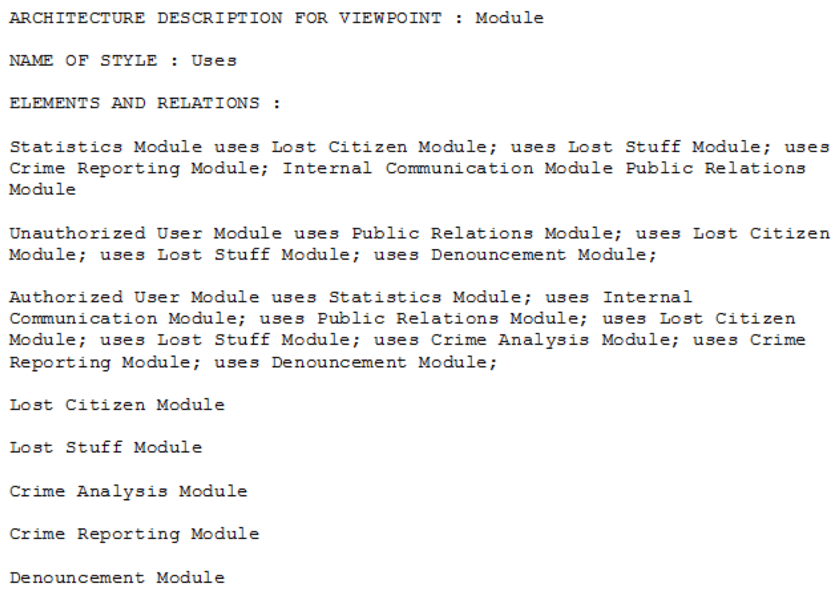
8.2. Example

We used Uses Style for Crime Management System Model (Figure 7) as an example and transform it to its textual representation for documentation. In Figure 14, template file of our transformation is seen and in Figure 15 textual representation of the model in Figure 7 is shown as an output of model-to-text transformation.



```
«IMPORT metamodel»

«EXTENSION template::GeneratorExtensions»

«DEFINE textDoc FOR Architecture»

    «FOREACH viewtypes AS v»
        «FOREACH v.styles AS s»
            «FILE s.styleName+".txt"»
                ARCHITECTURE DESCRIPTION FOR VIEWPOINT : «v.viewtype_name»

                NAME OF STYLE : «s.styleName»

                ELEMENTS AND RELATIONS :
                «FOREACH s.elements() AS e»
                «LET e.name.toString() AS currentElementName»
                «LET e.element_name.toString() AS currentElementType»
                «currentElementName» «currentElementType»
                «FOREACH s.relations() AS r»
                «IF r.source.name.toString().matches(currentElementName)»
                 «r.name» «r.target.name.toString()+" "+r.target.element_name.toString()»;
                «ENDIF»
                «ENDFOREACH»
                «ENDLET»
                «ENDLET»
                «ENDFOREACH»
            «ENDFILE»
        «ENDFOREACH»
    «ENDFOREACH»
«ENDDEFINE»
```

Figure 14. Textual Representation of Uses Style for Crime Management System



```
ARCHITECTURE DESCRIPTION FOR VIEWPOINT : Module

NAME OF STYLE : Uses

ELEMENTS AND RELATIONS :

Statistics Module uses Lost Citizen Module; uses Lost Stuff Module; uses
Crime Reporting Module; Internal Communication Module Public Relations
Module

Unauthorized User Module uses Public Relations Module; uses Lost Citizen
Module; uses Lost Stuff Module; uses Denouncement Module;

Authorized User Module uses Statistics Module; uses Internal
Communication Module; uses Public Relations Module; uses Lost Citizen
Module; uses Lost Stuff Module; uses Crime Analysis Module; uses Crime
Reporting Module; uses Denouncement Module;

Lost Citizen Module

Lost Stuff Module

Crime Analysis Module

Crime Reporting Module

Denouncement Module
```

Figure 15. Textual Representation of Uses Style for Crime Management System

9. Lessons Learned and Conclusions

In this section, we present the discussion of project development together with lessons learned. During the development of this work, we have gone through several experiences. Most importantly, we have examined the benefits and importance of MDSD. We started with domain analysis which is a tough activity. Then, we described the domain concepts in two ways: defining a grammar and defining a metamodel. Thus, we had the chance of comparing these two approaches. In addition to this, for metamodeling, we followed two procedures, one is from scratch and the other is UML profiling. We again had the opportunity of comparing these two approaches of metamodeling. We experienced with current tools for metamodeling. Lastly; we performed model-to-model and model-to-text transformations by using our predefined metamodel.

The first step of this work was to perform a detailed domain analysis and to identify domain concepts. We realized that performing a good analysis of the domain with considerable amount of time spent, and identifying the domain concepts clearly, helps building an accurate abstract syntax for the metamodel. Furthermore, it decreases the possibility of errors in the following phases.

The second step of the project was to define a DSL for the grammar. When we compare metamodel and grammar, we observe that metamodels are more understandable and suitable to express domain specific concepts. Grammar is not as expressive as metamodels for both visual and technical aspects. Relations between concepts are not defined clearly in grammars. In addition to this, in metamodel we can define constraints by using OCL however in grammar we cannot define constraints. We conclude that BNF is insufficient and is a harder way to express the relations between domain concepts.

As stated in the previous paragraph, metamodeling is a much more expressive and easy way of defining domain concept and their relations. In fact, during the development, we first created the abstract syntax of the metamodel and then we defined the grammar using this. In general, metamodeling based on MOF-from scratch is a difficult activity; however, it was not that difficult in our case since we had a small set of domain concepts. It still has a disadvantage for small cases, which is, it is not compatible with existing tools.

Next, we experienced with metamodeling based on UML profiling. For each concept in our domain, it was not hard to find a corresponding UML concept. So this approach is more efficient than from scratch case. This is also the case for concrete syntax definition. There is a well-defined concrete syntax for UML, thus we reused it mostly and made small adjustments for our case. There are also disadvantages that we faced with this approach. It does not give flexibility while defining the abstract syntax. Although this was not a big problem in our case, it may be more important in other domains.

For the static semantics, we used OCL for definition since UML is not enough. OCL is part of the UML standard and it is easy with it to define constraints for the metamodel which in turn increase the precision of models. One interesting experience we faced was the definition of topology concept in our domain. In reality, topology defines the constraints of a viewtype and its styles. We could define the constraints of viewtypes in static semantics with OCL. However, the constraints of styles which constitute topology reside in M1 level and thus cannot be defined in static semantics. By this observation, we conclude that topology itself corresponds to static semantics of each style defined and should be described by OCL. We did not define a seperate OCL for topology but just referred to it as a list of string values. One additional problem with topology was the difficulty of showing it in concrete syntax. We decided not to show it since it would complicate the model.

In the model-to model transformation part we used ATL that is flexible language for defining transformation rules and we transformed our model to an AADL model. Model-to-model transformation is complicated since there are endless variations of the source and target metamodels. Before starting implementation of the model-to-model transformation, transformation rules defined clearly and in detail. Thus, the source and target metamodel and semantics of transformation should be known. The most difficult part of the project is to analyze the internal structure of AADL metamodel in order to map our metamodel to AADL metamodel more accurately. In addition to this our metamodel and AADL metamodel are quite different and we faced problems about transforming the information contained in our model to AADL model.

In the model-to model transformation part we used ATL open-ArchirectureWare”. It is easier than model-to-model transformation. It is more understandable and simple and the tutorials of the tool are useful, although full coverage is not provided. We encountered one annoying problem is the untidy form of the output file. The document generated from transformation in Xpand is in an untidy form. To apply the correct indentation,

it requires hard work. So as to solve this problem, the syntax of Xpand should be reorganized and updated.
In overall, most of the tools developed for MDSD are still incubating (they are still under development), since MDSD is relatively a new concept. And because of this we suffered lack of documentation and advanced examples about tools and transformation languages.

In conclusion, in this paper we have defined a generic metamodel for defining architectural styles based on V&B approach. Our metamodel is simple and thus generic and extensible to use for various purposes. We have firstly defined a metamodel from scratch based on MOF. Although this is a heavyweight approach, we did not have much difficulty with it probably because of the simplicity of our metamodel. Next, we have defined a metamodel with UML profiling. UML profiling seems more favorable when tool support is considered. At the end of our experiment model-to-model and model-to-text transformation is applied. In model-to-model transformation we used AADL metamodel as target and in model-to-text transformation we transform our model to textual specification for documentation.

References

[1] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, September 2002.

[2] P. Kruchten. *The Rational Unified Process: An Introduction,Second Edition*. Addison-Wesley, Boston, MA, USA, 2000.

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

[4] P.H. Feiler, D.P. Gluch, J.J. Hudak.The Architecture Analysis and Design Language (AADL): An Introduction. Technical Note. CMU/SEI-2006-TN-011. February 2006.

[5] Eclipse Modeling Framework.
http://www.eclipse.org/modeling/emf/

[6] TOPCASED.
http://www.topcased.org/

[7] D. Garlan, J. Ivers, P. Clements, R. Nord, B. Schmerl and J. R. O. Silva. *Documenting Component and Connector Views with UML 2.0.* Technical report, CMU/SEI-2004-TR-008, Software Engineering Institute, 2004.

[8] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2007.