ELSEVIER

Available online at www.sciencedirect.com

The Journal of Systems and Software 79 (2006) 1180–1206

The Journal of Systems and Software

www.elsevier.com/locate/jss

AVDL: A highly adaptable architecture view description language

Jungwoo Ryoo [a], Hossein Saiedian [b,*]

[a] Information Sciences and Technology, Penn State Altoona, Altoona, PA 16601, United States
[b] Department of EECS, University of Kansas, Lawrence, KS 66045, United States

Received 1 November 2004; received in revised form 24 February 2006; accepted 25 February 2006
Available online 18 April 2006

Abstract

Architectural views are rapidly gaining a momentum as a vehicle to document and analyze software architectures. Despite their popularity, there is no dedicated language flexible enough to support the specifications of an unbound variety of views including those preexisting and needing to be newly created on demand. In this paper, we propose a novel view description language intended for specifying any arbitrary views, using a uniform set of conventions for constructing views and how to use them. The highly adaptable nature of the new language results from its built-in mechanisms to define different types of views in a systematic and repeatable manner.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Software architecture; Views; Viewpoints; Specifications; Meta object facility; Object constraint language; Extensible markup language; Schema; Meta-data interchange

1. Introduction

Software applications implementing non-trivial requirements are often so complex that they can be incomprehensible in their entirety. The IEEE recommended practice for architectural description (IEEE, 2000) defines software architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution". One can adopt this notion of architecture and alleviate both perceived and real complexity of software applications by describing them in terms of less granular and more abstract concepts such as components and their relationships. However, much of the original complexity often remains even in an architectural specification. To overcome this problem, many researchers (Perry and Wolf, 1992; Kruchten, 1995; Issarny et al., 1998; IEEE, 2000; Baragry and Reed, 2001; Clements et al., 2003; Wijnstra, 2003; Purhonen et al., 2004) advocate partitioning an architectural description into more than one view, which is "a representation of a whole system from the perspective of a related set of concerns (IEEE, 2000)". For example, inspired by the field of building architecture, Perry and Wolf state that "a software architect needs a number of different views of an architecture for the various uses and users (Perry and Wolf, 1992)".

Typically, there are *many* people and organizations interested in the successful construction of a software system. These are referred to as *stakeholders*. Examples of stakeholders include customers, end users, developers, customer care organizations, etc. Views not only reduce the complexity of architectural description but are also vehicles for separation of concerns by providing a highly specialized specification especially meaningful for a certain class of stakeholders.

The idea of views is further developed by the introduction of a higher-level abstraction referred to as *viewpoint*. "A viewpoint is a specification of conventions for constructing and using a view (IEEE, 2000)". In other words, it is a *pattern* or *template* from which to develop individual views by establishing the purposes and audiences for a view and the techniques

* Corresponding author. Tel.: +1 913 897 8515; fax: +1 913 897 8682.
E-mail addresses: jryoo@psu.edu (J. Ryoo), saiedian@eecs.ku.edu (H. Saiedian).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.02.045

for its creation and analysis. The idea of viewpoints is not new and has been used in various contexts of software engineering. For instance, in their requirements engineering research, Nuseibeh and Easterbrook (2000) use the term viewpoint to refer to "loosely coupled, locally managed, distributable objects capturing partial representational knowledge, development process knowledge, and specification knowledge about a system and its domain".

From these definitions one can conclude that (1) a view conforms to a viewpoint and that (2) a complete viewpoint specification requires, at a minimum, information about:

- the **process** governing the life cycle (creation, description, and analysis) of views instantiated from a viewpoint,
- the **style**: defining models and notations used by views belonging to a viewpoint, and
- the **domain**: describing the area of concerns addressed by a viewpoint and its instances.

Making a clear distinction between views and viewpoints is a desirable practice recommended by IEEE (2000). However, the term *viewpoint* is sometimes used interchangeably with *view* as in Kruchten (1995), Hofmeister et al. (1999), Herzum and Sims (1999). Clements et al. (2003) use the term *view style* in place of viewpoint.

Although solving the problem of complexity and satisfying the individualized concerns, the use of views poses its own set of challenges. One of the most significant among them is meeting the need for defining different types of views that may be either preexisting or needing to be created on-demand.

For instance, IEEE Std 1471-2000 (IEEE, 2000) provides basic requirements for views and viewpoints but falls short of establishing a consistent and uniform specification method.

Initially, researchers (especially, Kruchten, 1995) harbored a hope that there might exist an ultimate set of views addressing all the possible concerns of any arbitrary stakeholders. After trying these supposedly all-purpose views for real-life problems, practitioners are coming to a conclusion that no set of views are comprehensive enough to completely defy the necessity of introducing new kinds of views. The special requirements of unexplored domains prompted sporadic proposals (Purhonen et al., 2004) for a new set of highly specialized views, but the practice of describing views largely remains ad hoc.

Some of the consequences of this lack of a standardized and unified view specification mechanism include:

- difficulties in learning how to describe, discern, and refine well-known views due to the idiosyncrasies inherent to each different type of view,
- overhead associated with creating a new type of view since one must each time determine a specification method for it first, and
- incompatible view specifications that are not amenable to a formal analysis, which is due to employing disparate specification methods.

In this paper, we propose a novel language devoted to describing architectural views, which is deliberately designed to accommodate the need for defining and specifying different types of views in a systematic and repeatable manner. For the remainder of our discussion, the new language is referred to as AVDL (Architecture View Description Language). AVDL is based on an observation that a meta-relationship between any two views can be regarded as that of refinement. We show that the constraints governing the refinement relationships can be encoded into the metamodel of AVDL and hold the key to making it highly adaptable.

The organization of this paper is as follows. Section 2 lays the groundwork for our discussion of AVDL. Section 3 formally defines AVDL in terms of its metamodel and notations (both textual and graphical). Section 4 provides several concrete examples demonstrating how AVDL can be applied to a real-life problem. Section 5 discusses a case study we conducted to verify the practicality of our approach. We present our work in the context of related research in Section 6. Finally, we round out our paper with concluding remarks and further research.

2. Background

In order to understand the ensuing sections of this paper, it is critical to have some basic understanding on the languages and facilities such as UML, MOF, XML, and XMI. In this section we provide overviews on each of them.

2.1. UML

Unified modeling language (UML) (Object Management Group, 2003) is a visual modeling language and has a semi-formal syntax and semantics. It is intended to allow its users (mostly practitioners) to capture their design decisions in a group of easy-to-use diagrams when developing software applications. Each UML diagram addresses issues associated with a certain aspect of a software system under development. For example, a class diagram identifies classes (including their attributes and operations) and their relationships. A state diagram shows possible states in which a class can be

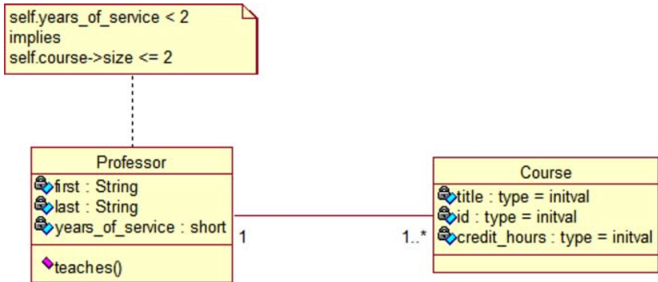

Fig. 1. A UML class diagram with an OCL constraint.

and how transitions happen from one state to the other. A sequence diagram visualizes the sequences of method invocations among instances of classes. There are many more UML diagrams than those introduced so far, and additional ones are being added to the original UML specification as needs arise.

Object constraint language (OCL) is a text-based formal declarative language developed by international business machine (IBM) to supplement the visual notations of UML. It is an extension to UML and provides a mechanism to specify more fine-grained constraints on UML models, which cannot be expressed by the original UML notations. OCL expressions also make it possible to query objects in UML models. For more in-depth information, readers are referred to the UML and OCL specifications (Object Management Group, 2003).

Shown in Fig. 1 is a UML class diagram. The boxes in the figure represent classes. The `Professor` class has attributes such as last name, first name, and years of service. It has one operation (namely, `teaches()`). The `Professor` class is associated with the `Course` class. In this relationship, the cardinality of the `Professor` class is one while the cardinality of the `Course` class is one to many. Although very expressive and intuitive in nature, the class diagram cannot specify more fine-grained constraints: for example, ‘‘if a professor’s years of service is less than two, then he or she teaches less than or equal to two courses per semester’’. Using OCL, one can easily document this constraint as shown at the top of Fig. 1.

2.2. MOF, XML, and XMI

In order to understand what MOF is, one must first grasp the concept of metamodel. Metamodel is a model defining a modeling language. For instance, UML has its own metamodel that unambiguously defines all the language constructs including classes, attributes, operations, associations, etc. Meta object facility (MOF) is a framework defining a language used to specify the metamodel of any modeling language (not tied to a specific modeling language) and is also referred to as a meta-metamodel. MOF is used to specify the metamodel of UML. MOF builds upon a modeling framework that is essentially a subset of the core UML specification (Object Management Group, 2002).

Extensible mark-up language (XML) is a World Wide Web Consortium (W3C) recommendation (Bray et al., 2000) for specifying a markup language. XML allows one to create his or her own markup language through the use of user-defined tags and the specification of their structural relationships in the form of a Document Type Definition (DTD) or Schema. Writing a DTD or Schema is equivalent to developing a new markup language.

XML metadata interchange (XMI) (Object Management Group, 2005) specifies how to create a corresponding XML DTD or Schema from MOF-compliant metamodels. The significance of XMI is that it enables one to transform the metamodel specification of a modeling language into a DTD or Schema. This transformation, in turn, generates a markup language that gives an XML notation for the modeling language. Using UML, Fig. 2 pictorially summarizes the relationships among MOF, XML, and XMI.

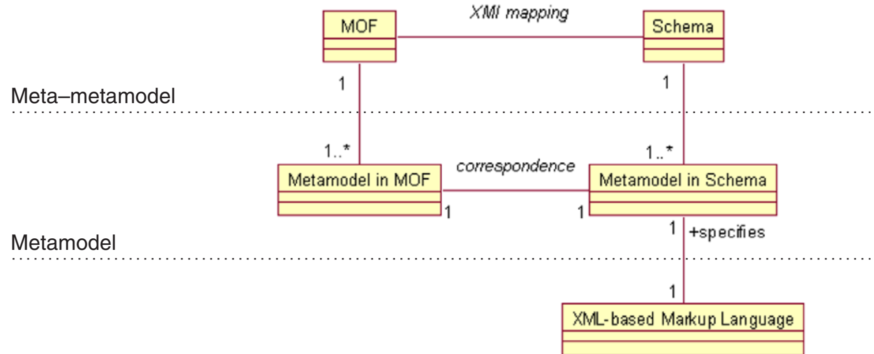

Fig. 2. The relationships among MOF, XML, and XMI.

For a concrete example, suppose that one wants a simple modeling language whose sole purpose is describing the detailed structure of a database table. The language would desirably have metamodel constructs like tables, columns, and composition relationships, which will be used to show the names and types of columns constituting a uniquely identifiable table. Fig. 3 utilizes MOF to depict the metamodel.

One can derive an equivalent XML Schema (Fig. 5) out of the MOF specification based on the XMI mapping between MOF and Schema.

Fig. 3. The metamodel of the database description language.

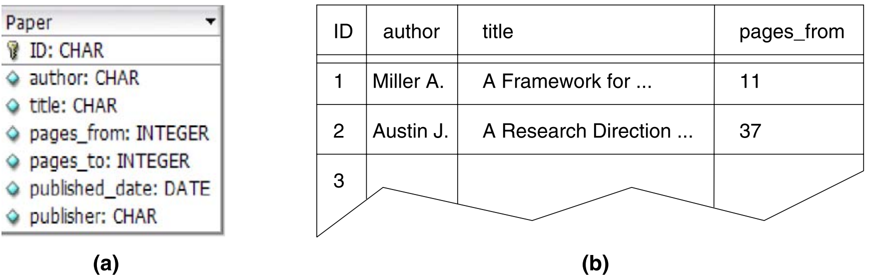

Fig. 4. A database table structure (a) and its instance (b).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:rdb="http:///rdb.ecore"
3              xmlns:xmi="http://www.omg.org/XMI"
4              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5              targetNamespace="http:///rdb.ecore">
6    <xsd:import schemaLocation="XMI.xsd"
7                namespace="http://www.omg.org/XMI"/>
8    <xsd:complexType name="Table">
9      <xsd:choice minOccurs="0" maxOccurs="unbounded">
10       <xsd:element name="Column" type="rdb:Column"/>
11       <xsd:element ref="xmi:Extension"/>
12     </xsd:choice>
13     <xsd:attribute ref="xmi:id"/>
14     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
15     <xsd:attribute name="name" type="xsd:string"/>
16   </xsd:complexType>
17   <xsd:element name="Table" type="rdb:Table"/>
18   <xsd:complexType name="Column">
19     <xsd:choice minOccurs="0" maxOccurs="unbounded">
20       <xsd:element ref="xmi:Extension"/>
21     </xsd:choice>
22     <xsd:attribute ref="xmi:id"/>
23     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
24     <xsd:attribute name="name" type="xsd:string"/>
25     <xsd:attribute name="type" type="xsd:string"/>
26   </xsd:complexType>
27   <xsd:element name="Column" type="rdb:Column"/>
28 </xsd:schema>
```

Fig. 5. A Schema equivalent to the metamodel in Fig. 3.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rdb:Table xmi:version="2.0"
3             xmlns:xmi="http://www.omg.org/XMI"
4             xmlns:rdb="http:///rdb.ecore" name="Paper">
5    <Column name="ID" type="CHAR"/>
6    <Column name="author" type="CHAR"/>
7    <Column name="title" type="CHAR"/>
8    <Column name="pages_from" type="INTEGER"/>
9    <Column name="pages_to" type="INTEGER"/>
10   <Column name="published_date" type="DATE"/>
11   <Column name="publisher" type="CHAR"/>
12 </rdb:Table>
```

Fig. 6. An XML document serializing Fig. 4(a).

With this Schema and its subsequent markup language, it is now possible to serialize the structure of a database table whose graphical notation is in Fig. 4(a). The consequence of the serialization is, in fact, an XML document (Fig. 6) marked up by the newly created language.

One of the important motivations for using standards such as MOF, XML, and XMI is the abundant availability of tools. For example, the Schema in Fig. 5 and the XML document in Fig. 6 are generated by one of such tools. Automation makes the conversion process (MOF → Schema → XML) less cumbersome and less error-prone.

3. AVDL: An architecture view description language

In this section the authors propose a novel mark-up language devoted to describing architectural views, which is deliberately designed to accommodate the need for defining different types of views and specifying their instances in a systematic and repeatable manner. For the remainder of the authors' discussion, the new language is referred to as AVDL (Architecture View Description Language). AVDL takes advantage of an observation that the meta-relationship between any two views can be regarded as that of refinement. This section shows that the constraints governing the refinement relationships can be encoded into the metamodel of AVDL and hold the key to making it highly scalable in depth and comprehensive in scope.

3.1. Architecture of AVDL

Modeling is a critical component of software engineering. Often a dedicated language is developed to aid modeling. Classical language theory provides an established way of creating such a modeling language, based on a layered architecture as shown in Fig. 7.

The information layer is composed of data that a modeling language is intended to describe. The model layer (using the modeling language) captures metadata describing data in the information layer. The metamodel layer is the embodiment of metadata defining the modeling language. The meta-metamodel layer specifies a language whose purpose is describing the

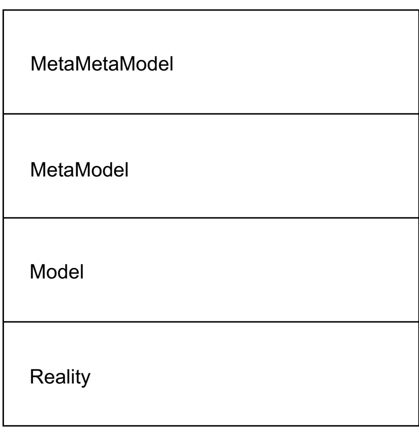

Fig. 7. The classical four layer metamodel architecture.

metamodel in the metamodel layer. For referential convenience, aliases are provided to refer to each layer of Fig. 7. For example, the information layer is dubbed as M0.

Theoretically, it is possible to infinitely expand the hierarchy of a modeling language (for example, meta-meta-meta-model), but the four-layered metamodel architecture depicted in Fig. 7 is the most commonly accepted one.

AVDL adopts the same layered architecture as the one shown in Fig. 7. In case of AVDL, the information layer is composed of data about a view that AVDL intends to describe. The model layer holds metadata (specified in AVDL) describing view data in the information layer. The metamodel layer contains the metadata defining AVDL (the language itself). The meta-metamodel layer specifies a language whose purpose is describing the AVDL metamodel.

Although metamodels are widely used in the creation of a modeling language (Karagiannis and Kuhn, 2002), few language-based software architecture specification methods have been developed through metamodeling. AVDL is deliberately designed to take full advantage of the classical four-layer metamodeling architecture discussed above. Instead of creating a new meta-metamodel, the authors choose to use an existing meta-metamodel. AVDL uses MOF (described in Section 2) as its meta-metamodel, which is possible because MOF is not a dedicated meta-metamodel framework belonging exclusively to UML. The benefits of adopting MOF are:

- its status as a de-facto industry standard meta-meta-model framework,
- its easy-to-learn nature for those already familiar with UML considering that it is a slimmed-down version of the UML specification, and
- its compatibility with Object Constraint Language (OCL) (Warmer and Anneke, 1998), eliminating the needs to search for another formalism to increase precision by adding more constraints to the specification of a metamodel.

3.2. AVDL metamodel

3.2.1. Overall design considerations

The metamodel decides the expressive power and language features of AVDL. An aspect of the expressive power the authors are especially interested in is the ability to describe an unbound set of view types and their instances, which is absent in the currently available view modeling approaches.

To achieve this goal, the authors use the concept of the *baseline* view type and *derived* view types, which results from their observation that there exists a view type, denoted as the baseline, addressing the essential architectural concerns shared across all the possible view types, serving as a common denominator. Therefore, the baseline view type shall be equipped with a core set of common constructs for any arbitrary type of view specification.

In the context of architecture refinement (Moriconi et al., 1995), the baseline view type is a template for the most primitive view with which any type of architectural modeling must begin. Every derived view type is considered to be the result of repeatedly refining either the baseline view type or other less refined view types through the application of one or more *constraints*. This automatically makes the views instantiated from the derived view types the refinement of those from the baseline or other view types located higher in the hierarchy. Fig. 8 is the depiction of the relationships between the baseline view type and derived view types. As shown in Fig. 8, there exists only one kind of relationship between view types: refinement.

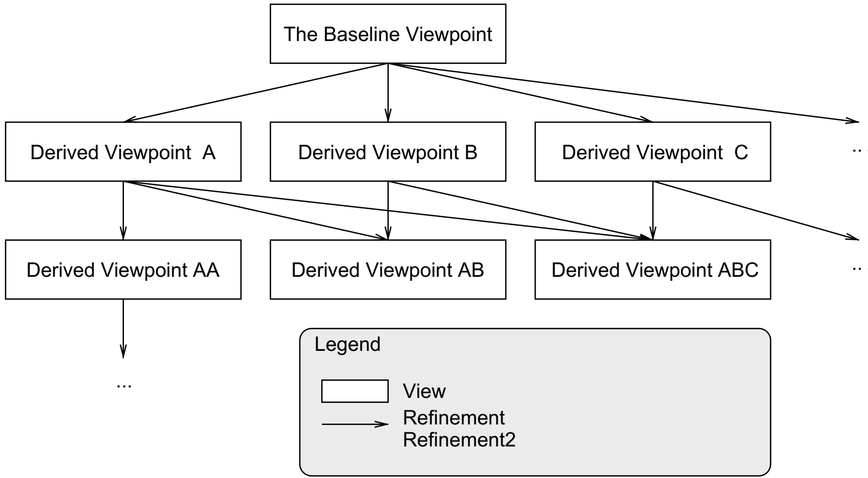

Fig. 8. The refinement-based taxonomy of views.

So how does the idea of the baseline view type and derived view types allow AVDL to accommodate unlimited number of view types? Initially, AVDL's metamodel is deliberately defined to only support the baseline view type. Next, the authors open up the metamodel to the users so that they can easily create an extension of AVDL, capable of specifying a new view type by applying additional constraints. UML takes a similar policy by disclosing its metamodel and providing multi-level extension mechanisms (such as constraints, tagged values, stereotypes, and profiles) that manipulate the metamodel.

Earlier the authors mentioned that MOF is the meta-metamodel of AVDL. One of the advantages of using MOF is that one can transform metamodels specified in MOF into extensible markup language (XML) (Bray et al., 2000) schemata (Thompson et al., 2001) using the predefined mappings (the OMG standard) between MOF and XML schema.

Using MOF makes it very straightforward to create a mark-up notation for AVDL since all one needs to do is defining AVDL by providing its metamodel. The way the authors envision how the baseline view type and derived view types should be handled in the context of the MOF-based AVDL metamodel is as follows:

- **Step 1:** Initially, AVDL's metamodel is shipped with the language constructs relevant to supporting only the baseline view type.
- **Step 2:** Using MOF, one generates an XML schema encapsulating the syntax necessary to describe the baseline view type, which, in turn, makes it possible to specify an instance of the baseline view type in XML.
- **Step 3:** As a need for a new view type surfaces, one revisits the original metamodel of AVDL (namely, the one dealing with the baseline view type only) and adds one or more constraints to it.
- **Step 4:** The refined metamodel (the original AVDL metamodel with newly added rules) then goes through the same schema generation process described in Step 2, resulting in an extended version of AVDL now capable of describing a derived view type.

One may further refine an already existing extension of AVDL by adding more constraints to its metamodel.

Having an XML-compliant canonical notation for AVDL is nice, but the usefulness of AVDL would not significantly exceed that of other more informal proposals (for example, 4 + 1 views and UML diagrams) employing dominantly graphical notations if there were not the access to the vast reservoir of related standards, tools, and extra capabilities that XML brings.

In the following sections, the authors will explore the details of the AVDL metamodel.

3.2.2. Primitives of the baseline view

The authors refer to the specification constructs demanded by the baseline view type as *primitives* since they represent a set of minimum ingredients necessary to produce any meaningful software architecture view specifications.

Closely examining one of the definitions of a *software architecture* sheds some light on what these primitives might be. Bass et al. (1998) state that:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

From the above definition, it is obvious that the primitives, at a minimum, must include elements, their relationships, and the behavior of these elements when participating in certain relationships. The nomenclature used here is intentionally generic because it is part of a definition. More technical terms for elements and relationships, widely accepted in the software architecture community are *components* and *connectors*.

The behavioral aspect of components is often built into the specification of connectors as demonstrated in the works of Allen (1997) and Allen and Garlan (1997). The authors adopt this approach and do not introduce a new primitive whose sole purpose is specifying the behaviors of components. Therefore, components and connectors are the two most fundamental primitives of the baseline view type.

Components represent the loci of computation. Connectors represent interactions among components. The degree of sophistication found in an interaction varies depending on the view type of which components are part. In a view type describing source code modules and their interdependencies, the role of a connector is simply indicating that one component *uses* another. However, a connector must be much more sophisticated when the view type involves runtime processes and concurrent interactions between them. Both components and connectors are either *atomic* or *composite*. Atomic components and connectors are not subject to any further architectural decomposition. Composite components and connectors are composed of smaller-grained sub-components and connectors.

Although necessary, components and connectors by themselves are not sufficient to fully specify an architectural view. For example, components and connectors have a potential to be part of an architectural topology, but they do not deliver the topology itself. The topology of an architectural view contains information about how components and connectors are combined to form a structure or structures. In this paper, the authors use the term *configuration* to refer to a primitive representing an architectural topology. Interfaces between components and connectors are to be represented by another

primitive denoted as a *port*. Ports are directional. They permit three types of interactions: *in*, *out*, and *inout*. Inout means that the port allows bidirectional interactions. In and out ports are unidirectional. That is, the in ports let only inbound actions occur while the opposite is true for the out ports. Components and connectors may have multiple ports. Fig. 9 shows how components, connectors, ports, and a configuration can specify the topology of an architectural view. Note that the configuration information is implicitly specified in terms of the pictorial arrangement of components, connectors, and ports. The same topological information can be also explicitly specified in a canonical format.

The legend for the notations used in Fig. 9 is provided in Table 1.

3.2.3. Derivatives of the baseline view type

The primitives in Section 3.2.2 do not offer sufficient *hooks* to which one can apply the constraints necessary to build an AVDL extension that is capable of specifying instances of derived view types. To overcome this lack of room for derivation, AVDL introduces both *local* and *global* attributes to its primitives. The distinction between global and local attributes is based on whether they are the attributes of every AVDL primitive or not.

Each view records a snapshot of the architect's decisions affecting one or more stakeholders. A stakeholder represents a group of people taking part in a unique set of software engineering activities called *disciplines* (Robillard and Kruchten, 2003). Examples of these disciplines are requirements elicitation, analysis and design, implementation, testing, etc.

Depending on which discipline to illuminate, a view specification requires totally orthogonal types of constraints that need to be imposed upon the same primitive despite some common ground. For instance, a component in the context of the analysis and design discipline is merely an abstract, conceptual entity that may or may not exist in reality while it takes a more concrete form such as source code in the implementation discipline. By the time the same component reaches the testing discipline it becomes executable and is treated accordingly as a process. This metamorphic nature of the architectural elements and their relationships must be factored into developing the specification constructs of the baseline view.

The discipline-specific language features associated with the primitives of AVDL must be manifested by the application of orthogonal constraints that can be collapsed into the following four major categories (Baragry and Reed, 2001):

- logical:static,
- logical:dynamic,
- physical:static, and
- physical:dynamic.

The logical elements are defined as the conceptual (existing only in the imagination of stakeholders) abstractions of a desired system structure and its behavior. On the contrary, the physical elements are the abstractions of the tangible implementations of the logical elements. They are produced during the implementation discipline and directly traceable to concrete objects in the real world, such as source code snippets, application servers, self-contained software components, etc.

The static elements of a view distill only the topology of an architecture while their dynamic counterparts represent the run-time behavior projected on top of the static elements.

To handle this specialization (especially in the case where the specialization adds more *constraints* only), one can introduce a new property to every primitive proposed in Section 3.2.2, which reveals its true nature as being logical:static, logical:dynamic, physical:static, or physical:dynamic. For referential convenience throughout this paper, the authors denote such a property as a *view context*. One thing to consider here is the presence of primitives that are not subject to further

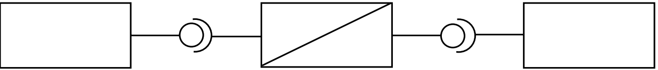

Fig. 9. An architectural topology specified in terms of components, connectors, ports, and a configuration.

Table 1
The notations in Fig. 9 and their meanings

Notation	Meaning
	Component
	Connector
	Port for an outbound action
	Port for an inbound action

categorization and do not demonstrate any different properties in any of the four view contexts. Configuration and Port fit this description and are referred to as being *neutral*.

Once the view context of a primitive is determined, one can add extra constraints to the AVDL primitives in addition to those required by the view context itself. This leads to the creation of a new derived view type.

The straightforward approach of using the view context property of the AVDL primitives and extra constraints imposed on them fails when additional specification constructs are required to further elaborate on an existing primitive and to supply more points of extension. As briefly discussed earlier in Section 3.2.2, connectors dealing with concurrent interactions among runtime processes are highly complex and must be equipped with more features than those needed by static connectors. In fact, connectors whose view context is static just need to specify that there are associations between components at the baseline view type level. The actual types of these associations are defined in a derived view type with additional constraints.

The new features needed to support the dynamism in connectors can be summarized as the capability to describe a collection of events occurring in a certain sequence between components via a connector. Therefore, at least three new specification constructs are imperative to support a dynamic connector:

- **events** representing a unique occurrence triggered by the action or reaction of a component, resulting in a state change or changes in components linked to the same connector,
- **orderings** representing a sequence in which events are happening, and
- **behavioral traits** representing the actual behavior of a dynamic connector, consisting of events and orderings.

Instead of referring to these new constructs as primitives, the authors call them *derivatives* because they are a new set of specification constructs whose primary function is providing additional features to the existing connector primitive in the baseline view type.

3.2.4. AVDL metamodel specification in MOF

Now the stage is set to more rigorously define the metamodel of AVDL. The primitives and derivatives identified in Sections 3.2.2 and 3.2.3 are the core set of the mandatory building blocks of specifying any arbitrary view types of a software architecture. AVDL fully incorporates these constructs into its metamodel, but does so with some significant adjustments and optimizations. For instance, instead of repeatedly specifying the identical properties in its constructs, the AVDL metamodel factors out the commonality and starts with the lowest common denominator. The metamodel's most abstract member is *Model Element* as shown in Fig. 10.

The *view_contextKind* type of the view_context attribute in *ModelElement* simply enumerates the possible view contexts of any given architectural components or connectors. As a result, the value of the *view_contextKind* type can be one of {logical:static, logical:dynamic, physical:static, physical:dynamic, and neutral}. Every construct in the metamodel also has a distinct identifier called name, which is reflected in the *name* attribute of *ModelElement*.

Shown also in Fig. 10 are the basic metamodel constructs upon which other more prominent constructs (such as components and connectors) are built. These are constructs neutral to any specific view context except for *Event* and *BehavioralTrait* whose view contexts are always dynamic.

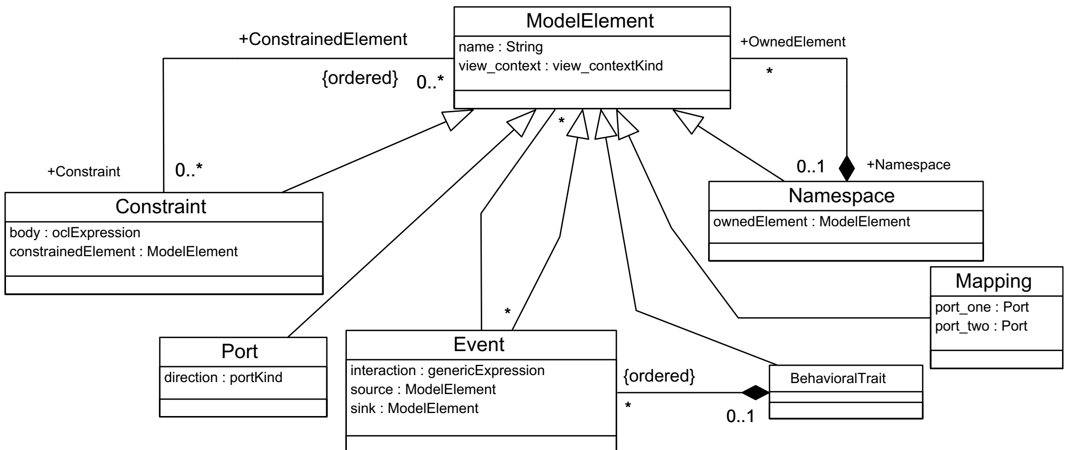

Fig. 10. Basic AVDL metamodel constructs.

- **Constraint** is indispensable for extending AVDL so that it can specify the individual instances of derived view types. In AVDL it is the main mechanism of extension allowing the specification of the offspring of the baseline view type. In fact, the baseline view type is not associated with any constraints at all while a derived view type has at least one constraint attached to itself. The relationship between Constrint and ModelElement in Fig. 10 shows that any model element in a derived view can be constrained. The oclExpression type used as the type of a constraint body implies that object constraint language (OCL) (Warmer and Anneke, 1998) is the language used to specify actual constraints in AVDL. OCL is part of the UML specification and can be used to clarify the semantics of MOF-based metamodels.
- **Namespace** enables one to create composite components and connectors. The relationship between Namespace and the rest of the constructs in Fig. 10 is recursive and follows the well know *Composite* design pattern (Gamma et al., 1994).
- **Mapping** is a construct required to define the Configuration construct appearing in Fig. 13. It has the *port_one* and *port_two* attributes that specify the connection between two ports originating from disparate components or connectors. For example, the port_one attribute of a mapping may be initialized with the name of a port belonging to a component while the port_two attribute value is set to the port name of a connector. In addition to ModelElement, Mapping is another new concept peculiar to the AVDL metamodel and does not have its counterpart in the collection of the primitives and derivatives discussed in Sections 3.2.2 and 3.2.3. The addition of these newly identified concepts in the AVDL metamodel is mainly due to the increased specificity necessary in transforming those primitives and derivatives into the metamodel constructs.
- **Event** represents the action or reaction of components involved in the specification of a connector whose view context is dynamic as discussed in Section 3.2.3. The *source* attribute of Event contains a component triggering the event. *Sink* represents a component affected by the event.
- **Behavioral Trait** represents an aggregation composed of the instances of Event. Using the {ordered} feature of MOF eliminates the need of creating a separate metamodel construct for the *ordering* primitive discussed in Section 3.2.3.
- **Port** is equivalent to the port primitive defined in Section 3.2.2. It has the direction attribute whose value is one of {in, out, inout}.

With these basic building blocks, one can now define a component as shown in Fig. 11. Due to the fact that the behavior of a component is specified as part of a connector, the metamodel specification for Component is relatively simple.

Defining a connector is more complicated because it involves additional constructs such as *Event* and *BehavioralTrait*. As mentioned in Section 3.2.3, only a connector in a dynamic view context is permitted to have a behavioral trait. Therefore, the relationship between BehavioralTrait and Connector is aggregation instead of being composition. Fig. 12 shows Connector defined in MOF.

Finally, Fig. 13 defines *Configuration*. The authors realize that there are three different types of configuration:

- **NestedComponent** representing a composite component with its own configuration consisting of sub-components, sub-connectors, etc.,
- **NestedConnector** representing a composite connector with its own configuration consisting of sub-components, sub-connectors, etc., and,
- **Model** representing either a simple configuration consisting of non-nested components and connectors, or a composite configuration containing any potential combinations of NestedComponent, NestedConnector, non-nested constructs.

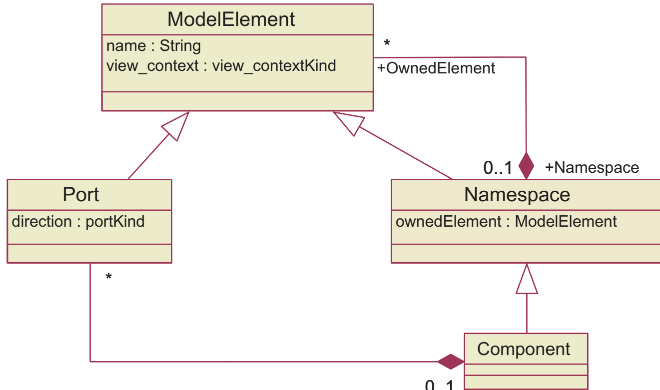

Fig. 11. Component defined in MOF.

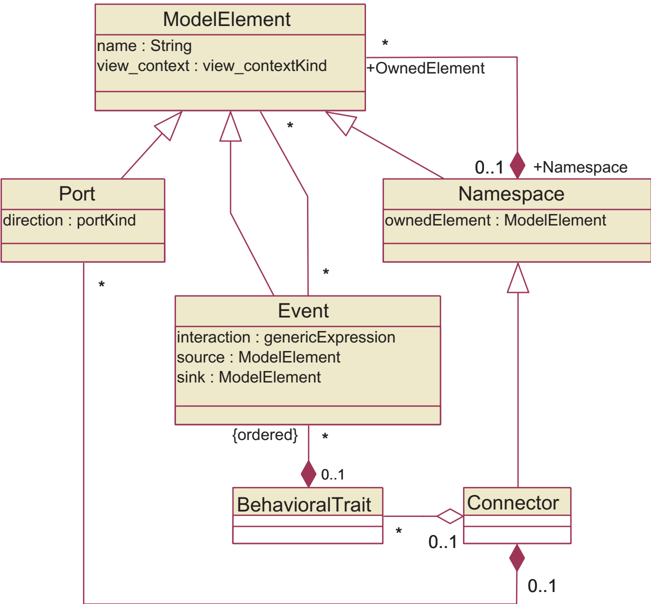

Fig. 12. Connector defined in MOF.

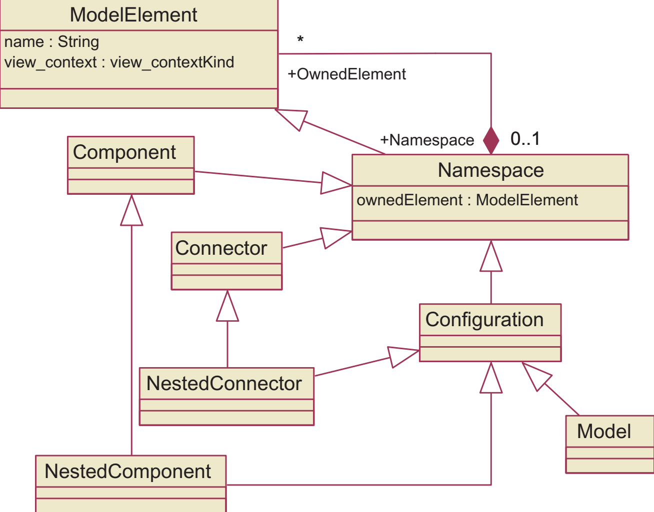

Fig. 13. Configuration defined in MOF.

Inheriting from Namespace (due to its recursive definition) makes it possible for these three types of Configuration to own any arbitrary model elements including components, connectors, ports, etc. as well as lower level NestedComponent, NestedConnector, and Model.

3.3. Textual notation for AVDL

Taking advantage of the OMG's XMI specification (Section 2), one can generate an XML schema (Fig. 16) from the AVDL metamodel in Fig. 10.

The textual notation for AVDL is a well-formed XML document that is validated successfully against the schema in Fig. 16.

As explained in Section 3.2.1, to specify a view conforming to a derived view type, one must use an extension of AVDL obtained by constraining the original AVDL metamodel. Although there are many formal and semi-formal ways to specify constraints, the authors recommend the Object Constraint Language (OCL) (Warmer and Anneke, 1998) for this task mainly because OCL is a language recommended by OMG and specifically designed for writing constraints over UML and MOF models.

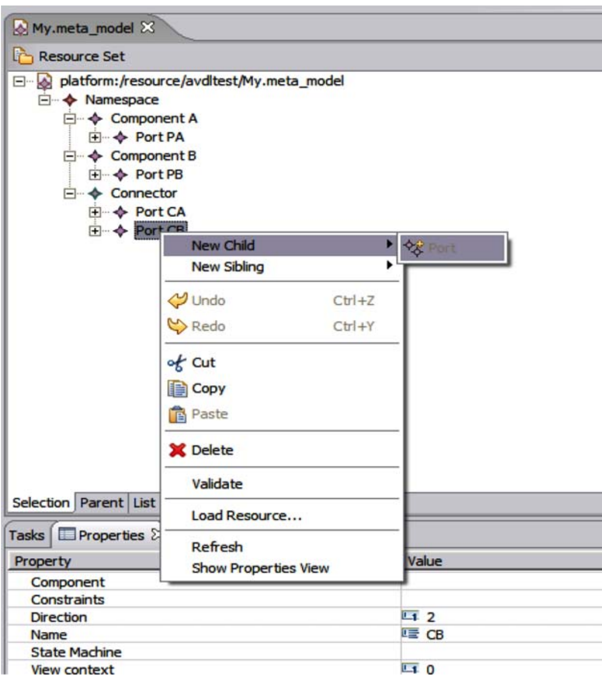

Fig. 14. A GUI-based context-sensitive editor for AVDL.

```
1  avdlCmpType : enum {module}
2  avdlCntType : enum {uses}
3
4  Component->forAll(self.view_context = logical:static or
5                    physical:static)
6  Connector->forAll(self.view_context = logical:static or
7                    physical:static)
8
9  Component->forAll(oclType = module)
10 Connector->forAll(oclType = uses)
11
12 Connector->forAll(c | c.port.allInstances->size = 2)
13
14 Connector.port->forAll(p1, p2 | p1.direction = in implies
15                                 p2.direction = out)
16
17 Component->forAll(p1, p2 | Connector->forAll(n1, n2 |
18                   p1.port.mapped_to = n1.port and
19                   p2.port.mapped_to = n2.port and
20                   n1 = n2 implies
21                   p1 <> p2))
22
23 Component->forAll(p1, p2 | Connector->forAll(n1, n2, n3, n4 |
24                   p1.port.mapped_to = n1.port and
25                   p2.port.mapped_to = n2.port and
26                   p1.port.mapped_to = n3.port and
27                   p2.port.mapped_to = n4.port and
28                   n1 = n2 and n3 = n4 implies
29                   n1 = n2 = n3 = n4))
30
```

Fig. 15. OCL constraints defining a derived view type.

Therefore, describing an instance of a derived view type in AVDL involves:

- a **pre-condition**: defining the derived view type in terms of OCL constraints imposed upon the AVDL metamodel or an already existing derived view type definition,
- a **specification**: creating an XML document specifying an instance of the view type, and
- a **post-condition**: verifying the integrity of the specification (desirably using an automated tool).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xmi="http://www.omg.org/XMI"
3  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4  xmlns:meta_model="http:///meta_model.ecore"
5  targetNamespace="http:///meta_model.ecore">
6    <xsd:import schemaLocation="XMI.xsd"
7  namespace="http://www.omg.org/XMI"/>
8    <xsd:complexType name="ModelElement">
9      <xsd:choice minOccurs="0" maxOccurs="unbounded">
10       <xsd:element name="Event"
11 type="meta_model:Event"/>
12       <xsd:element name="Constraints"
13 type="meta_model:Constraint"/>
14       <xsd:element ref="xmi:Extension"/>
15
16 ...
17
18     </xsd:choice>
19     <xsd:attribute ref="xmi:id"/>
20     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
21     <xsd:attribute name="name" type="xsd:string"/>
22     <xsd:attribute name="view_context" type="xsd:string"/>
23     <xsd:attribute name="Event" type="xsd:string"/>
24     <xsd:attribute name="Constraints" type="xsd:string"/>
25
26 ...
27
28   </xsd:complexType>
29   <xsd:element name="ModelElement"
30 type="meta_model:ModelElement"/>
31   <xsd:complexType name="Event">
32     <xsd:complexContent>
33       <xsd:extension base="meta_model:ModelElement">
34         <xsd:choice minOccurs="0" maxOccurs="unbounded">
35           <xsd:element name="ModelElement"
36 type="meta_model:ModelElement"/>
37         </xsd:choice>
38         <xsd:attribute name="ModelElement"
39 type="xsd:string"/>
40       </xsd:extension>
41     </xsd:complexContent>
42   </xsd:complexType>
43   <xsd:element name="Event"
44 type="meta_model:Event"/>
45
46   ...
47
```

Fig. 16. The AVDL metamodel expressed as an XML schema.

For example, one can define a view type by applying the OCL constraints in Fig. 15 to the AVDL metamodel. The OCL expressions state that: (1) the *view_context* of every component in the view type is either logical:static or physical:static, (2) the *view_context* of every connector is either logical:static or physical:static, (3) all the components are of *module* type, (4) all the connectors are of *uses* type, (5) every connector has two ports, (6) the direction of one port belonging to a connector is *in* while direction of the other is always *out*, (7) a connector cannot have a *uses* relationship with itself, and (8) there can be only one connection between two components.

At the baseline view type level, a Graphical User Interface (GUI)-based context-sensitive editor can completely remove any possibility of composing an invalid XML document containing an AVDL model since the tool enforces the rules of the AVDL metamodel and prevents a user from making a mistake as shown in Fig. 14.

However, the same editor cannot guarantee the validity of a view instantiating a derived view type due to the additional OCL constraints. For this purpose, a separate OCL parser and a mechanism to interrogate a view specification against the corresponding OCL-based view type definition is necessary.

4. Example

In this section, we put AVDL to work to describe the architectural views of a small but real-life software system. We take our example from the domain of network alarm management. Assume that there exist two legacy systems. One collects simple network management protocol (SNMP) traps from network elements. SNMP (Mauro and Schmidt, 2001) is an application layer protocol facilitating the exchange of management information between network devices. The other system translates raw alarm information and displays the results through a graphical user interface (GUI) implementation. There is no direct connectivity between the two systems. Our example system intercepts traps from the collection system and forwards them to the display system.

Following the AVDL process discussed in Section 3.2.1, the first thing we do is developing the baseline architectural view of the new system. We identify components and connectors required in the baseline view as

- **interceptor**, which is a component responsible for intercepting the SNMP traps from their source (namely, the collection system),
- **distributor**, which is a component responsible for distributing alarms to one or more instances of the display system, and
- **transporter**, which is a connector responsible for transporting the alarms from the interceptor to the distributor.

Using the notations in Table 1, we show the graphical representation of the baseline view in Fig. 17.

The same view can be specified in an XML document as shown below.

```
<Component>
  <name>Interceptor</name>
  <Port>
    <name>icrptl</name>
    <direction>out</direction>
  </Port>
<Component/>

< Configuration>
   <Mapping>
     <port_one>icrptl</port_one>

    <port_two>ctrptl</port_two>
  </Mapping>
  ...
```

```
<Connector>
  <name>Transporter</name>
  <Port>
    <name>ctrptl</name>
    <direction>in</direction>
  </Port>
  <Port>
    <name>ctrpt2</name>
    <direction>out</direction>
  </Port>
</Connector>
```

Later in the design process, more refined architectures emerge. Among them is an approach using a combination of Perl scripts, a C executable, and named pipes. Perl (Schwartz and Phoenix, 2001) is an acronym for *Practical Extraction and*

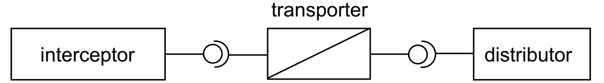

Fig. 17. The baseline view of the example system.

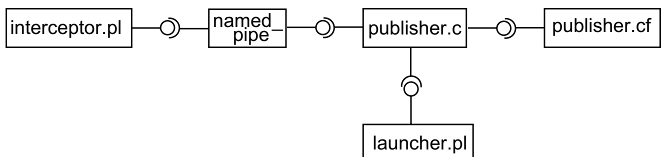

Fig. 18. The *uses* view in a static view context.

Report Language and highly popular due to its powerful text-manipulation functions. A named pipe (Stevens, 1990) is a UNIX interprocess communication mechanism. When an SNMP trap arrives, the event triggers the execution of a Perl script which, in turn, writes the details of the trap to a named pipe. Another Perl script spawns a process running a C executable designed to periodically read the content of the named pipe and publish the SNMP trap details to an Internet communication port.

This architecture is first captured in a view depicting the source code modules and their static dependencies as shown in Fig. 18. Note that we use a *shorthand* notation here and do not explicitly show any connectors in this scenario to simplify the graphical specification. In this view, components are software modules while connectors represent a *uses* relationship in which the correct operation of one module requires the existence of the other. Therefore, the net differences between the view in Fig. 18 and the views instantiated from the baseline viewpoint in Fig. 17 are the additional constraints to the AVDL metamodel expressed in OCL below.

Component
```
self->forAll(view_context=static)
```

Connector
```
self->forAll(view_context=static)
self->forAll(name="uses")
self->forAll(port->size=2)
self.port->forAll(p1, p2 |
p1.direction=in implies p2.direction=out)
```

`interceptor.pl` (Fig. 18) is the refinement of the interceptor component in Fig. 17. The named pipe corresponds to the transporter but is regarded as a component in this view although the transporter is identified as a connector in the baseline view. This is acceptable because the classification of architectural elements may change depending on which view they are part of. `launcher.pl` and `publisher.c` are derived from the distributor component. `launcher.cf` is a configuration file dictating the behavior of both `launcher.pl` and `publisher.c`.

```
<xsd:simpleType name="view_context">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="static"/>
  <xsd:restriction>
</xsd:simpleType>
```

Shown above are net changes in the schema corresponding to the component portion of the modified AVDL metamodel representing a viewpoint in Fig. 18 assuming that the original AVDL schema (for the baseline view) is what is shown below.

```
<xsd:element name="name" type="xsd:string"/>
<xsd:simpleType name="direction">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="in"/>
    <xsd:enumeration value="out"/>
    <xsd:enumeration value="inout"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:simpleType name="view_context">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="static"/>
    <xsd:enumeration value="dynamic"/>
    <xsd:enumeration value="logical"/>
    <xsd:enumeration value="neutral"/>
  <xsd:restriction>
</xsd:simpleType>
```

```
<xsd:element name="Port">
  <xsd:element ref="name"/>
```

```
  <xsd:element ref="direction"/>
</xsd:element>

<xsd:element name="Component">
  <xsd:element ref="name"/>
  <xsd:element ref="view_context">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Port"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

One may decide to slightly change the architecture by adding to `publisher.c` a new feature to launch itself, effectively making a separate launcher (launcher.pl) unnecessary (Fig. 19). With this change, our example architecture is now qualified to be specified using the conventions of a viewpoint denoted as *layered*.

In the layered viewpoint, a component:

- is not allowed to use another component when it is already used by the component to be used,
- is linked to two or less connectors, and
- is used by another component or uses another component.

The *layered* viewpoint is more formally and succinctly defined by adding the following OCL constraints to the *uses* viewpoint.

Namespace

```
component->forAll(c1, c2 |
(configuration.mapping.port_one
= c1.port.name and
configuration.mapping.port_two
= c2.port.name and
c1.name > c2.name)) implies
c1.port.direction = out and
c2.port.direction = in
```

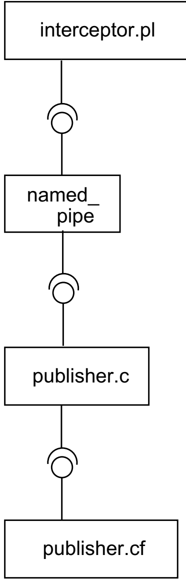

Fig. 19. The *layered* view in a static view context.

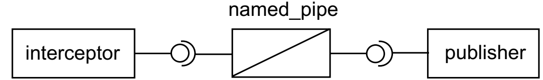

Fig. 20. The pipe-and-filter view in a dynamic view context.

In addition to the static viewpoint instances in Figs. 18 and 19, another specifies the run-time behaviors of the same system (Fig. 20). Notice that the components in this view represent processes during the execution time. The dynamic aspect of the architecture is emphasized by an explicit connector that could be further scrutinized in terms of *events* and *behavioral traits*.

The events the stakeholders are especially interested in are those of interceptor writing to the named pipe and publisher reading from the named pipe. More specifically, they want to avoid premature reading by the publisher when writing by the interceptor is taking place. Reading and writing must always occur at two different times. This unique behavioral property of the connector can be distilled in a viewpoint enforcing the following OCL constraints added to the baseline viewpoint:

Component
```
self->forAll(view_context = dynamic)
```

Connector
```
self->forAll(view_context = dynamic)
self->forAll(name = "named_pipe")
self->forAll(port->size = 2)
event_type : enum {write, read}
self.behavioraltrait.event->
forAll(e.event_type = write xor e.event_type = read)
status : enum {active, inactive}
self.port->forAll(p1, p2 |
p1.status = active implies p2.status = inactive)
```

Configuration
```
self->exist(c | c.component.port.direction = out implies
c.connector.port.direction = in)
self->exist(c | c.component.port.direction = in implies
c.connector.port.direction = out)
```

5. Case study

We conducted a case study to test whether AVDL provides a practical and scalable solution to specifying the architectural views of a real-life, industrial-strength software system. We were able to develop a number of new viewpoints to help the process of architectural design as shown in Fig. 21. Some of the viewpoints are domain-specific redefinitions of well-known viewpoints while others are completely new.

Although practioners agree on the importance of developing a solid architecture early in the design process, they often have a vague idea on how to approach an architectural design. During the case study, we discovered that the use of AVDL is highly helpful in making the concept of software architecture concrete and tangible. With the aid of AVDL, a software architecture is no longer an abstract idea but a tangible process of modeling and specifying views in AVDL.

We also realize that AVDL helps enforce the philosophy of architectural refinement. Since the viewpoints are defined in a refinement hierarchy, instantiating viewpoints by specifying views ensures that a view specification is a refinement of another.

The case study is based on a large-scale network alarm management system (NAMS) whose functional requirements include:

- Software agents are deployed to network devices, monitor their statuses, and generate network alarms according to a predefined set of rules.
- Alarms are first sent to a predesignated local repository with an interface that is capable of forwarding them to a different location.
- The alarms are transformed into application-specific messages, further processed, and then presented to multiple graphical user interface (GUI) clients.

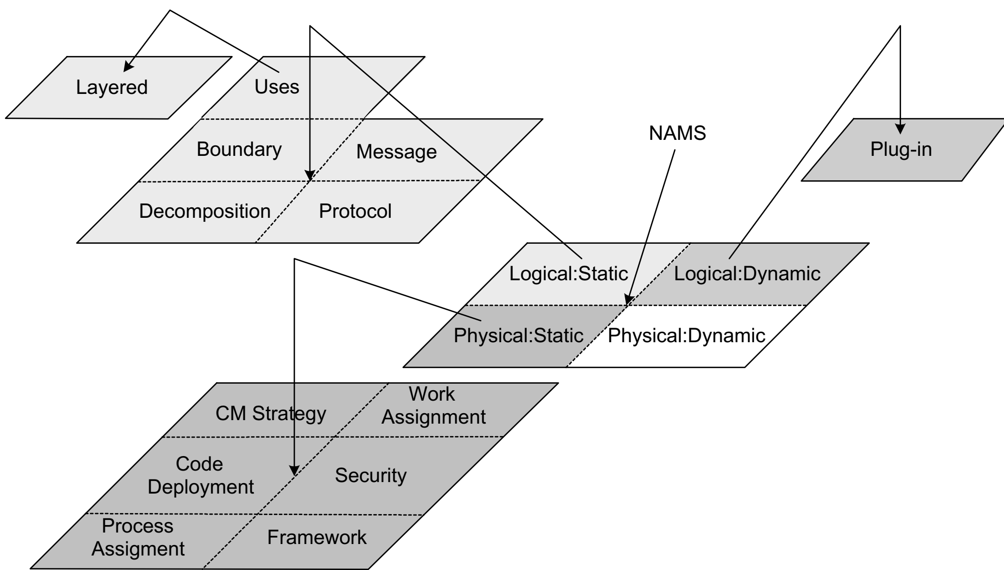

Fig. 21. The view types used for the architectural design of NAMS.

A large network consists of numerous network devices. The stability of the network depends on how closely these devices are monitored so that the number of malfunctioning devices is minimized. All the network devices generate alarms, and the network managers must be able to only see relevant alarms indicating potential problems while suppressing unimportant alarms. The most critical role of NAMS is providing its users with this filtering capability.

The project used for the case study involved more than ten thousand lines of source code. Due to the space limitation, this section only presents the details of the logical:static viewpoint and its derived viewpoints.

5.1. The logical:static views of NAMS

What decides the semantics of the logical:static viewpoint is the definition of *logical* and *static*. The AVDL metamodel specification formally defines *logical* as the first value (which can only be either logical or physical) of an ordered pair that decides the value of the *view_context* of a *ModelElement* whose instances must exist in the logical:static or logical:dynamic viewpoint. On the other hand, *physical* is the first value of an ordered pair that decides the value of the *view_context* of a *ModelElement* whose instances must exist in the physical:static or physical:dynamic viewpoint.

In the NAMS projects, a more tangible definition of the term logical is used. The team maintains a list of physical entities used for the project, which includes the source code, configuration files, start-up scripts, directories, processes, libraries, configuration management facilities, virtual machines, compilers, schedulers, hosts, firewalls, etc. The first value of an ordered pair that decides the value of the *view_context* of a *ModelElement* is physical if the name of the *ModelElement* appears in the NAMS list of physical entities. Otherwise, it is logical.

Static is the second value of the ordered pair that decides the value (which can only be either static or dynamic) of the *view_context* of *ModelElement* that is either a connector without a *BehavioralTrait* or component connected to a connector without a *BehavioralTrait*.

One can succinctly define the logical:static viewpoint as

Viewpoint Logical:static

```
Component
self->forAll(view_context = logical:static)

Connector
self->forAll(view_context = logical:static)
```

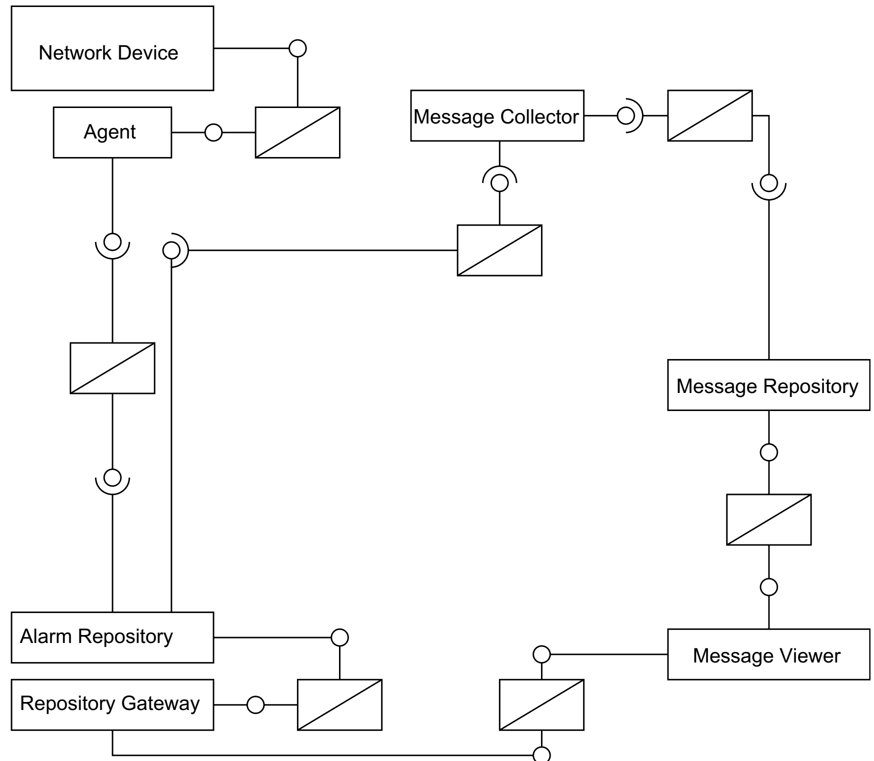

Fig. 22. Logical:static view of NAMS.

Shown in Fig. 22 is a graphical representation of the logical:static view of NAMS.

The view reveals a highly primitive structure consisting of all the major components and connectors with very little information on their architectural implications. It simply shows that the components have relationships manifested by connectors assuming (as in any other modeling languages) that its users understand the basic syntax and semantics of AVDL as well as the view-specific, extra, explicit constraints imposed on the AVDL's metamodel. In fact, this simple nature of the logical:static viewpoint is deliberately engineered so that the stakeholders can build its instances as brain-storming exercises with minimum restrictions and information on an intended architecture of a system during relatively early stages of an architectural design process.The plain logical:static view can be freely refined by the stakeholders to reflect their own specific needs. For example, the NAMS development team (in charge of implementing the system by producing code deliverables) wanted to have a view showing a clear boundary between the system to be built and the other external systems that will be interacting with it. The new viewpoint (we refer to it as the boundary viewpoint) is defined as shown below.

Viewpoint Boundary

Stereotype `BndryComponent`

```
--1-- The view_context property of a BndryComponent is always logical:static.
l[em] self->forAll(view_context = logical:static) and
--2-- A BndryComponent has a value indicating its membership (the domain expert decides
whether a component or connector is part of the new system or not).

bndryMbrs: enum {internal, external}
```

Stereotype `BndryConnector`

```
--1-- The view_context property of a BndryConnector is always logical:static.
self->forAll(view_context = logical:static) and

--2-- A BndryConnector has a value indicating its membership (the domain expert decides
whether a component or connector is part of the new system or not).

bndryMbrs: enum {internal, external}
```

```
Component
self->forAll(com | com.stereotype = BndryComponent)

Connector
self->forAll(con | con.stereotype = BndryConnector)
```

Note that the specification above can be rewritten in the form of the logical:static viewpoint definition $+\triangle$ since the boundary viewpoint is a refinement of the logical:static viewpoint. If presented this way, only the additional constraints (namely, $\triangle$) need to be reviewed by those who are already familiar with the AVDL syntax and semantics relevant to the logical:static viewpoint.

In the case of NAMS, a database table was created to keep track of the membership information (regarding whether a component or connector is external or not) of all the known components and connectors. Fig. 23 is a graphical representation of the boundary view of NAMS. Components and connectors in gray indicate that they are external to NAMS.

One of the most common concerns among the stakeholders is how network alarms flow through NAMS and what transformation rules are applied to them from entry to exit. This view turns out to be especially beneficial to the testers and maintainers (such as customer care or production support personnel) of the system since the view unambiguously shows the preconditions and postconditions for alarm processing in the context of the overall, desired movements of alarms.

For instance, one of the test cases assigned to a tester might be verifying whether an alarm group that initially looks like one shown in Fig. 24(a) is properly transformed into Fig. 24(b). Note that a message group is a reformatted version of an alarm group. One of the possible reformatting options is changing the sequence of the alarms in the alarm group. For a customer care specialist, it is critical to understand the expected message processing behavior of NAMS and to quickly identify a potential source of a problem being reported, and the message flow view proved to be a very useful tool for this purpose.

We define the message view as shown below.

Viewpoint Message
Stereotype `MsgPort`
`--1-- MsgPort has a value characterizing itself as a source or sink of an alarm.`

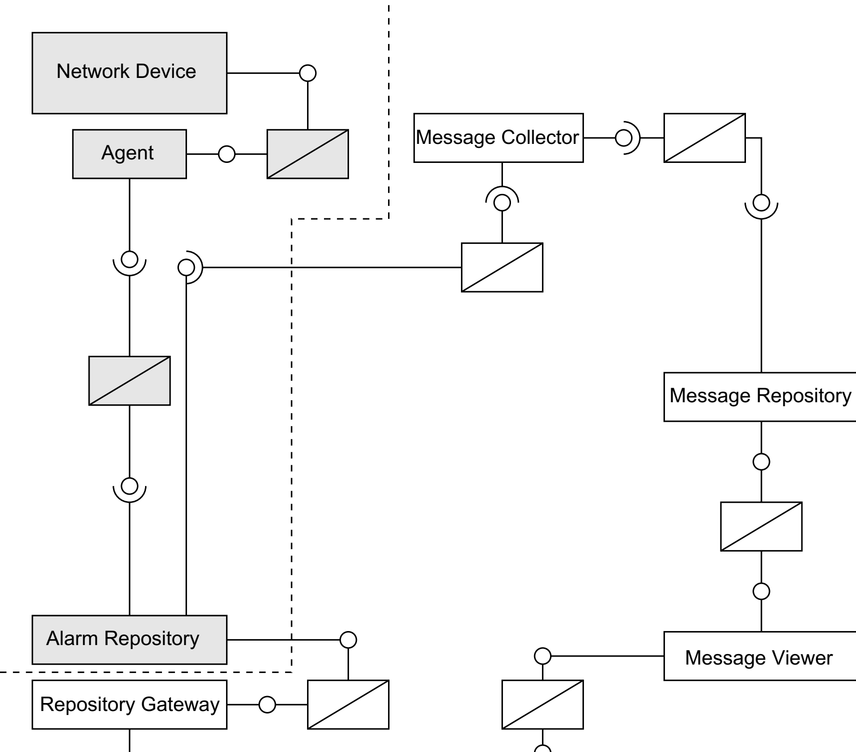

Fig. 23. The boundary view of NAMS.

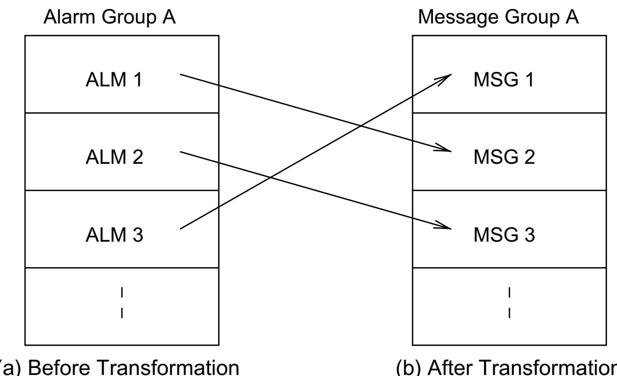

Fig. 24. Alarm transformation rules.

```
msgRole: enum {source, sink}

--2-- The direction property of MsgPort is either in or out.

self.direction->forAll(d | d <> inout)

--3-- The direction property of MsgPort is always in when its msgRole property is set to
sink.

self->forAll(direction = in implies msgRole = sink)

--4-- The direction property of MsgPort is always out when its msgRole property is set to
source.

self->forAll(direction = out implies msgRole = source)
```

Stereotype MsgComponent

```
--1--The view_context property of MsgComponent is always logical:static.

self->forAll(view_context = logical:static)

--2--All MsgComponent ports must be of stereotype MsgPort.

self.Port->forAll(p | p.stereotype = MsgPort)

--3--MsgComponent has a value indicating whether the transformation of an alarm occurs in
itself.

msgIsTransformed: enum {yes, no}

--4--MsgComponent has a value storing the message transformation logic.

msgTransLogic: schema

--5--When the msgIsTransformed property of MsgComponent is set to no, msgTransLogic must
be null.

self->forAll(msgIsTransformed = no implies msgTransLogic = null)
```

Stereotype MsgConnector

```
--1--The view_context property of MsgConnector is always logical:static.

self->forAll(view_context = logical:static)
```

```
--2--The name property of a MsgConnector is always sends_message_to.

self->forAll(name = sends_message_to).

Component
self->forAll(com | com.stereotype = MsgComponent)

Connector
self->forAll(con | con.stereotype = MsgConnector)
```

An XML document conforming to the schema of *msgTransLogic* is also shown below.

```
<alm_grp name = A>
  <alm seq_no = 1>
    <name_bfr>fault_counter_alm</name_bfr>
    <name_afr>fault_counter_msg</name_afr>
    <data_type>integer</data_type>
  </alm>
  <alm seq_no = 2>
  ...
</alm_grp>
<alm_grp name = B>
...
```

The graphical representation of the NAMS message view is shown in Fig. 25.

The component in which message transformation occurs is highlighted in gray. When users click on the component, a GUI screen appears and allows them to view the details of the transformation logic by providing an ability to navigate alarm groups along with its associated alarms.

As the architectural design of NAMS matures, the existing components are decomposed into more fine grained ones. By default, AVDL can describe this decomposition process without imposing any extra constraints to its original metamodel. Therefore, for the decomposition view in Fig. 26, no new viewpoint definition is necessary.

The connectors now have names representing the protocols used between components. For this additional feature, one needs the definition shown below.

```
Viewpoint Protocol

Stereotype PrtclComponent
--1-- The view_context property of a PrtclComponent is always logical:static.
self->forAll(view_context = logical:static)

Stereotype PrtclConnector
--1-- The view_context property of a PrtclConnector is always logical:static.

self->forAll(view_context = logical:static) and

--2-- The PrtclConnector has a value representing a collection of all the possible
protocols for NAMS.

Prtcl: enum {SNMP, J2EE, JNI, RMI, JDBC} and

--3-- The name property of the PrtclConnector always has a value belonging to the
collection Prtcl.

self->forAll(name = Prtcl->select())
```

```
Component
self->forAll(com | com.stereotype = PrtclComponent)

Connector
self->forAll(con | con.stereotype = PrtclConnector)
```

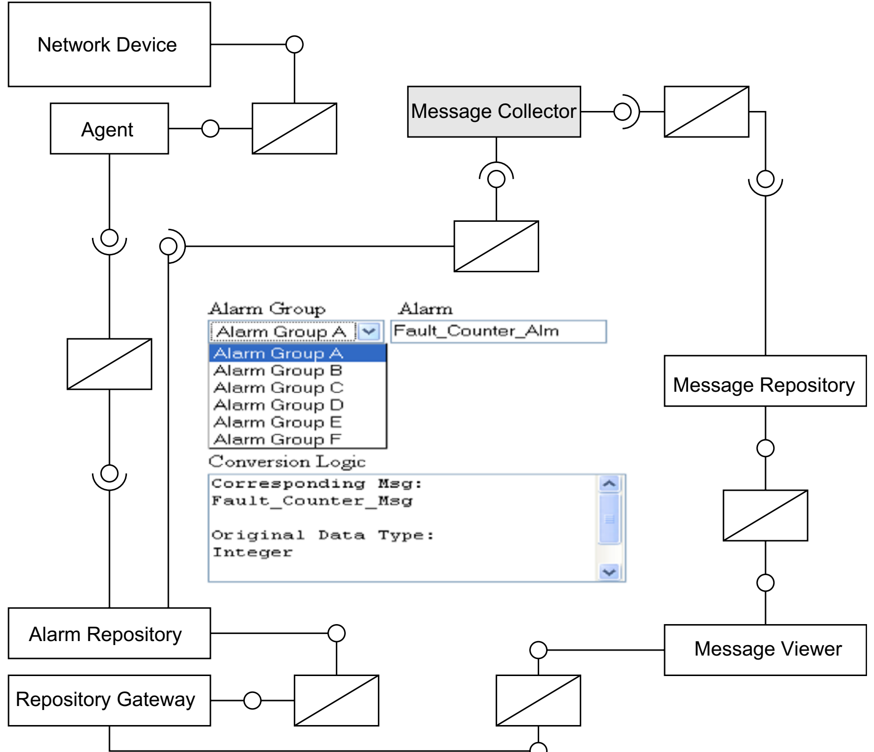

Fig. 25. The NAMS message view.

6. Related work

6.1. Current use of views in architectural description

Kruchten (1995) is one of the earliest advocates of adopting views in architectural description. He proposes 4 + 1 views (Fig. 27).

In his porosal, the *logical* view is the object model. The *process* view captures the concurrency and synchronization aspect of a software architecture. The *physical* view describes the mapping(s) of the software onto the hardware. The *development* view presents the static organization of the software in its development environment. The fifth view is a scenario view that provides a common context for the other views.

Since the introduction of 4 + 1 viewpoints, other researchers (Hofmeister et al., 1999; Herzum and Sims, 1999) have also proposed their own. Some viewpoints from one proposal can be easily mapped onto those from another while others cannot as shown in Table 2. Viewpoints comparable to one another are placed in the same row of Table 2. The mismatches (the cells containing × in Table 2) are mainly due to the differences in concerns being handled, the scope, and the abstraction level and imply that it is not feasible to develop a set of predefined views covering all the possible domains. For instance, the viewpoints of Herzum and Sims are so much more abstract than those of Kruchten and Hofmeister that the given mappings barely make sense. After all, the mappings themselves are subjective and liable to change when done by different people.

Although valuable in their own way, the viewpoints discussed so far are, at times, too broad and abstract to be truly helpful to a certain group of stakeholders. Realizing this, Clements et al. (2003) develop a more comprehensive hierarchy

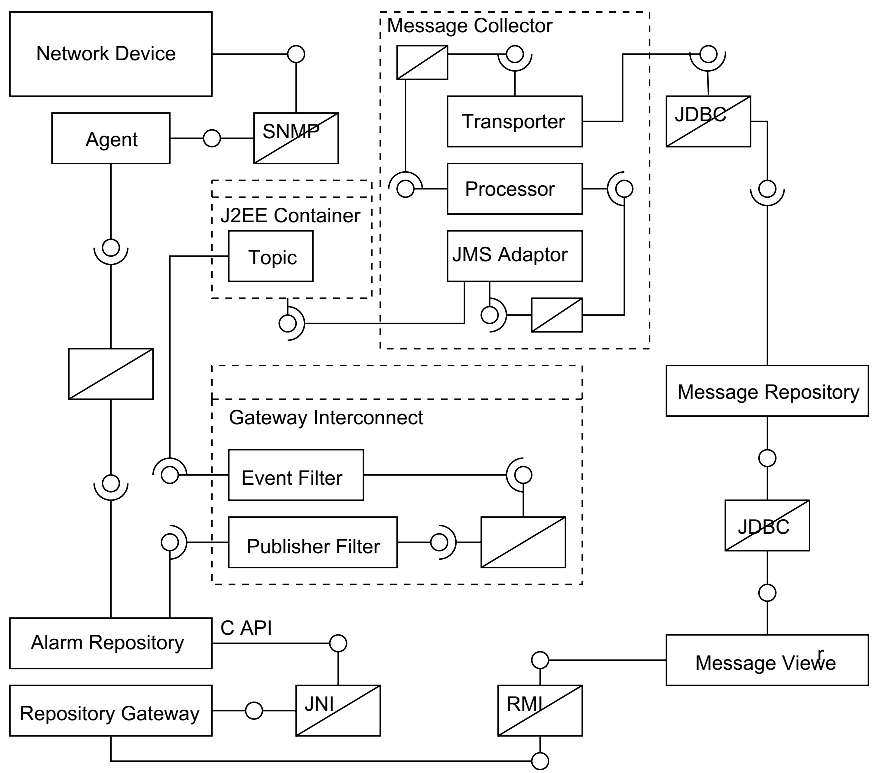

Fig. 26. The protocol view.

of viewpoints in both scope and granularity. They identify *module*, *component-and-connector*, and *allocation* as three major groupings at least one of which any arbitrary viewpoint must belong to. Each major category is then further refined into concrete viewpoints customized to best describe a certain architectural style as illustrated in Fig. 28. An architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined (Shaw and Garlan, 1996).

Clements et al.'s work is an extension of the previous research on views in the following respects:

- accepting the possibility of infinitely many viewpoints by understanding their open-ended nature,[1]
- avoiding the rigid partitioning of view space into mutually exclusive viewpoints concentrating on the unequivocally broad and abstract aspects, and
- recognizing architectural styles as patterns significant enough to warrant independent viewpoints.

Despite being more mature than any of its predecessors, Clements et al.'s viewpoint collection still exhibits some deficiencies of its own. For instance, it decouples itself from any specific means of capturing architectural views. This strategy was originally intended to avoid tying a viewpoint to a certain specification method but also results in a situation in which a unique method can potentially exist for each different viewpoint.

In theory, the diversity of choices is supposed to be an advantage since they allow more precise and specialized treatment of a viewpoint. In reality, however, the approach is not very practical because stakeholders would have to learn all the idiosyncrasies of as many methods as the number of viewpoints they want to use. Even worse is difficulty in achieving any reasonable level of integration (traceability and consistency in particular) between views belonging to different viewpoints due to their incompatibility to each other.

On the other end of the spectrum, researchers (especially, Kruchten with his 4 + 1 views) have tried to use a single general-purpose modeling language such as Unified Modeling Language (UML) (Booch et al., 1998) for specifying multiple viewpoints. Although highly successful from the perspective of popularization, this *one-fits-all* solution suffers from a problem of the lack of precision and specialization.

In the following section, the authors look more closely into currently available view specification methods in relation to AVDL.

[1] Each individual stakeholder may potentially demand a unique view.

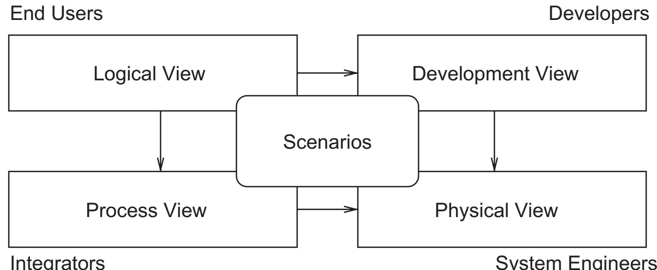

Fig. 27. Kruchten's 4 + 1 views.

Table 2
Viewpoint mappings among different approaches

Kruchten	Hofmeister et al.	Herzum and Sims
Logical	Conceptual	Application
Process	Execution	×
Development	Module interconnection	Functional
Development	Code view	Functional
Physical	Execution	Technical
×	×	Project management

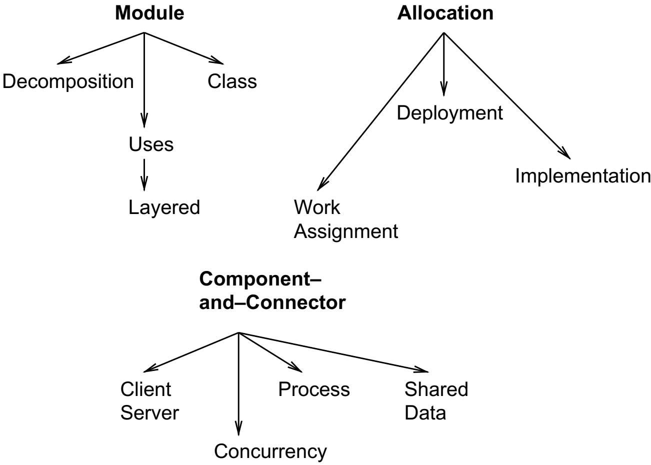

Fig. 28. Clements et al.'s viewpoints.

6.2. AVDL and other view specification methods

With good infrastructural support one can greatly improve the efficiency of his or her modeling efforts. Typically, such an infrastructure consists of specification mechanisms, processes, and tools (Fig. 29), which are referred to as the triad of successful modeling in this paper.

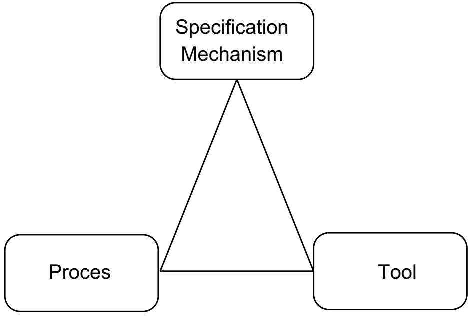

Fig. 29. Triad of successful modeling.

Most of the existing modeling techniques concentrate on one of them, especially the specification mechanisms. For example, some use a pure formal method like Z. This is completely feasible but inefficient. The inefficiency stems from the lack of predefined building blocks specifically designed to describe views since the formal methods are meant to be *general purpose*. As a result, architectural view specifications based on pure formal methods are ad hoc (built every time from scratch for a particular view) in nature. An alternative is finding a model (originally designed for other purposes), borrowing its constructs, and using them for describing views. Although more effective, this approach still imposes on its users the burden of redefining the semantics of the model's constructs.

The best of all is using a dedicated language fully featured to document and help analyze views. Unified Modeling Language (UML) is touted as *the language* for this since Kruchten's $4+1$ views (Kruchten, 1995) first appeared. A subset of UML models, including class, component, and deployment views, are identified as those directly supporting Kruchten's views. Viewpoints addressed by these UML models are limited in scope and expandability (what if one needs to create a new view beyond the capabilities of the three UML models?), and therefore not suitable for the concrete coverage of more specialized architectural views.

Medvidovic et al. (2002) observe these deficiencies and propose constraining the metamodel of UML to extend the language further into the domain of architectural description. The intended goal of their research is establishing UML extensions capable of describing the types of views supported by any arbitrary architecture description language (ADL) (Medvidovic and Taylor, 2000) such as C2 (Medvidovic et al., 1997), Wright (Allen, 1997), and Rapide (Luckham et al., 1995).

An ADL is a suite of solutions for representing and analyzing software architectures. Its role is not limited to just providing formal modeling notations. Supporting multiple views is considered one of many required features of an ADL.

Note that Medvidovic et al. do not directly tackle the UML's lack of comprehensive support for more concrete and focused viewpoints. Rather, they are interested in using UML as an ADL. In the context of this paper, we realize that the significance of Medvidovic et al.'s work lies in overcoming UML's limited support for views by opening a possibility to accommodate unrestricted number of new viewpoints. Their approach to constraining the metamodel to produce the desired syntax and semantics is a *portable* technique applicable to not only UML but also any other languages, and AVDL does take advantage of it.

Creating a UML extension for each viewpoint required is possible but not practical especially considering the overhead of redefining the non-trivial UML metamodel repeatedly and enforcing the new syntax and semantics in the existing tools. Furthermore, Medvidovic et al. do not address the problem of view integration at all. AVDL is optimized for view description. Its metamodel is streamlined and composed of constructs only relevant to describing architectural views, which is a key to achieving efficiency and automation.

Lastly, Medvidovic et al. fall short of providing a consistent and standardized method. In fact, it heavily relies on existing architectural description mechanisms to create corresponding UML extensions. Different UML models and their metamodels have to be manipulated depending on what description mechanism is being modeled after.

In addition to this ad hoc nature, a more serious problem arises when one invents a new viewpoint that cannot be associated with any existing specification methods. The implication is that he or she must invent a completely new dedicated language and then create a UML extension for it.

7. Concluding remarks and further research

An architectural view is a vehicle to reduce the complexity of an architectural specification. It also plays a role of a filter presenting only relevant information to the right target audience, which is necessary because different categories of stakeholders in a software project have different information needs.

In this paper we have explored the use of a new language called AVDL as a core element to our strategy to tackle many challenges involved in specifying viewpoints and views. We believe that solving the immediate problem of viewpoint/view specification will eventually lead us to a breakthrough in resolving a much bigger problem of building a comprehensive framework to manage the entire life cycle of architectural views. For example, the framework we envision will have additional processes and tools to retrieve and manipulate the existing views on the fly.

AVDL taps into the well-established industry standards such as MOF, XML, and XMI. OMG provides specifications for MOF and XMI. XML is a W3C standard. We use MOF to define the metamodel of AVDL. XMI allows us to serialize the models conforming to the AVDL metamodel specification in MOF by providing the mappings between MOF and XML schemata. Therefore, the output of the AVDL model serialization process is an XML document distilling a view of a software architecture.

The ability to serialize AVDL models in XML documents is not very useful by itself, but when accompanied by a strong theory and a detailed process, it becomes a powerful means of creating a new viewpoint.

From the perspective of end users who will use AVDL to actually describe views, using the XML-based notation may be a demanding experience even with the help of an advanced XML editor since XML is more geared toward machine

consumption. To improve the readability of the AVDL notation for human consumption, we are currently developing a graphical notation for AVDL.

Finally, measuring the effectiveness of AVDL against other known view specification methods is what we think is necessary but not attempted in this paper. We plan to develop a metric to quantify the effectiveness of a view specification.

References

Allen, R.J., 1997. A formal approach to software architecture. PhD Thesis, School of Computer Science, Carnegie Mellon Universit.

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6 (3), 213–249.

Baragry, J., Reed, K., 2001. Why we need a different view of software architecture. In: The Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE Computer Society, Amsterdam, the Netherlands, pp. 125–134.

Bass, L., Clements, P., Kazman, R., 1998. Software Architecture in Practice. Addison-Wesley, Reading, MA, ISBN 0-201-19930-0.

Booch, G., Jacobson, I., Rumbaugh, J., 1998. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA.

Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., 2000. Extensible Markup Language (XML) 1.0, second ed. W3C Recommendation, October 2000.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2003. Documenting Software Architectures. Addison-Wesley, Reading, MA, ISBN 0-201-70372-6.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Herzum, P., Sims, O., 1999. Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons, New York, ISBN 0471327603.

Hofmeister, C., Nord, R.L., Soni, D., 1999. Describing software architecture with UML. In: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX. International Federation for Information Processing. pp. 145–160.

IEEE, 2000. IEEE recommended practice for architecture description. IEEE Standard 1471.

Issarny, V., Saridakis, T., Zarras, A., 1998. Multi-view description of software architectures. In: Proceedings of the 3rd International Software Architecture Workshop, Orlando, FL, USA, November 1998. pp. 81–84.

Karagiannis, D., Kuhn, H., 2002. Metamodelling platforms. In: Proceedings of the 3rd International Conference on EC-Web 2002 – Dexa 2002, Aix-en-Provence, France, September 2002. Springer-Verlag, Berlin, pp. 182–195.

Kruchten, P., 1995. The 4 + 1 view model of architecture. IEEE Software 12 (6), 42–50.

Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W., 1995. Specification and analysis of system architecture using rapide. IEEE Transactions on Software Engineering 21 (4), 336–355.

Mauro, D.R., Schmidt, K.J., 2001. Essential SNMP. O'Reilly.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.

Medvidovic, N., Oreizy, P., Taylor, R.N., 1997. Reuse of off-the-shelf components in C2-style architectures. In: Proceedings of the 19th International Conference on Software Engineering, Boston, MA. IEEE Computer Society, Silver Spring, MD, pp. 692–700.

Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E., 2002. Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology 11 (1), 2–57.

Moriconi, M., Qian, X., Riemenschneider, R.A., 1995. Correct architecture refinement. IEEE Transactions on Software Engineering 21 (4), 356–372.

Nuseibeh, B., Easterbrook, S., 2000. Requirements engineering: A roadmap. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland. ACM, pp. 35–46.

Object Management Group (OMG), 2002. Meta Object Facility (MOF) specification, Version 1.4, April 2002.

Object Management Group, 2003. OMG unified modeling language specification, Version 1.5, March 2003.

Object Management Group, 2005. XML Metadata Interchange (XMI), v2.1, OMG Specification, September 2005.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40–52.

Purhonen, A., Niemelá, E., Matinlassi, M., 2004. Viewpoints of dsp software and service architectures. Journal of Systems and Software 69 (1–2), 57–73.

Robillard, P.N., Kruchten, P., 2003. Software Engineering Process with the UPEDU. Addison-Wesley, Reading, MA.

Schwartz, R.L., Phoenix, T., 2001. Learning Perl, third ed. O'Reilly.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspective on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-131-82957-2.

Stevens, W.R., 1990. UNIX Network Programming. Prentice-Hall, Englewood Cliffs, NJ.

Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N., 2001. XML Schema part 1: Structures. W3C Recommendation 2, May 2001.

Warmer, J., Kleppe, A., 1998. The Object Constraint Language: Precise Modeling with UML, first ed. Addison-Wesley Publishing Company, Reading, MA.

Wijnstra, J.G., 2003. From problem to solution with quality attributes and design aspects. Journal of Systems and Software 66 (3), 199–211.