Validating UML Models Against Architectural Profiles

Petri Selonen
Tampere University of Technology
Institute of Software Systems
P.O. Box 553
FIN-33101 Tampere, Finland
petri.selonen@tut.fi

Jianli Xu
Nokia Research Center
P.O. Box 407
FIN-00045 Helsinki, Finland
jianli.xu@nokia.com

ABSTRACT

The Unified Modeling Language (UML) has become a widely adopted standard in the software industry. While UML has established itself in detailed software design, its usage as an architecture description language is still taking its shape. In particular, there is a growing need for techniques to define domain specific architectural constraints and conventions in UML. We address this issue by adopting the concept of UML profiles for architectural design. Architectural profiles are specialized for describing and constraining software architecture descriptions for a given domain. We argue that these profiles represent an appropriate abstraction level to elaborate architectural constraints and conventions. We present a general schema for arranging architectural profiles and a set of conformance rules that define how these profiles are interpreted, constituting a profile definition language for validating architectural design. We introduce a tool for performing architectural validation and discuss the results of our initial case studies.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies, tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*CASE, object-oriented design methods*

General Terms

Design, Documentation

Keywords

UML, profiles, architectural validation

1. INTRODUCTION

Designing and describing software models using UML [9] is a common practice in the software industry. UML is increasingly used for architectural modeling as well: as a general-purpose standard design notation, UML is a credible alternative to dedicated architecture description languages (ADLs). UML descriptions of software architecture not only provide a standardized definition of system structure and terminology, but also facilitate a more consistent and broader understanding of the architecture, and enable more extensive tool support for architecture design. It is relatively easy to proceed from a UML architecture description into detailed system and component design, where UML and OO programming paradigms are regularly used.

However, there is a lack of methodological support for using UML for describing software architecture. Both our experiences with software development practices at Nokia [11], and the research of others [8, 3], suggest that there is an urgent need of effective strategies and tools for supporting software architects working with UML. One requirement is the possibility of defining domain, product line, or platform specific architectural conventions guiding the architects and allowing conformance testing of architectural designs.

Medvidovic et al. [8] assessed three different strategies for modeling software architectures with UML: (1) Use UML "as is", (2) customize the UML metamodel using UML's built-in extension mechanisms, and (3) extend the UML metamodel to directly support the needed architectural concepts. Because of the crucial requirement for standardization in industrial software development, we adopt the second strategy. It has a better capability of describing software architecture compared to the first one; as it involves standard UML, it can reap the benefits of existing UML tools, in contrast to the third one.

In the approach described in this paper, *architectural profiles* play a key role in materializing the work context of software architects. Architectural profiles are UML profiles that are specialized for software architecture design process and software architecture description. Generally, architectural profiles determine the subset of UML used for architecture description, as well as the necessary extensions for modeling architectural entities. More importantly, architectural profiles define the structural and behavioral constraints and rules of the architecture under design.

Architectural profiles can be used to drive, check, and automate the software architecture design process and the creation of all architectural views. We expect that architectural profiles are particularly helpful for enforcing the conventions of product-line architectures. Architectural profiles should be followed in the process of architecture design in order to guarantee that the resulting architecture design has necessary properties and lacks undesirable ones. We believe that architectural profiles are on an appropriate abstraction level to elaborate architectural constraints.

In order to make use of architectural profiles in the way discussed above, a practical technique to specify profiles in UML is required. Unfortunately, presently UML itself gives no guidelines on how to do this. Due to various possible views on software architecture, and different purposes that profiles can be used for, the profiles can take different forms. In this paper we propose a technique for specifying profiles within UML, aiming at a specification and tool-supported enforcement of architectural constraints. In particular, we show how profile specifications can be structured according to different views on software architecture, and how a subset of UML can be used as a profile specification language with a precise interpretation. The interpretation is expressed as a set of rules, each imposing a particular type of conformance requirement on the model. An advantage of this approach is that the designer can select a desired set of conformance rules (or even a single rule) to be checked in the model.

UML CASE tools (e.g. Rational Rose [10]) typically focus on OO design and programming, and consequently on low-level details and code generation. They can help the architects to edit a UML model, check syntax of the model, and manage the model to a certain degree. However, these tools provide no means for checking whether a given UML model conforms to certain architectural conventions and constraints, specified as profiles. Since manual checking is inefficient and error-prone, the success of a profile-centric approach relies on adequate tool support. We present a tool that performs model validation based on architectural profile specifications.

We have applied the profile-based architectural validation approach presented here to the validation of an industrial system architecture based on a mobile device software platform. The results have been positive in the sense that we have been able to point out actual, previously undetected conflicts with the conventions of the platform architecture.

The paper is structured as follows. In Section 2, we present the general principles of UML architectural profiles together with a chosen model structure and profile hierarchy. In Section 3, we define a set of validation rules that define the profile language used in the paper. Section 4 gives an example of validating architectural views against profiles. In Section 5, we introduce the tool implementation. In Section 6, we discuss our case study. Section 7 presents related work. Finally, in Section 8 we give concluding remarks.

2. UML ARCHITECTURAL PROFILES AND MODEL STRUCTURE

2.1 General

The UML Model Management package [9] (pp. 2-187 to 2-204) specifies how model elements are organized into models, packages, subsystems, and UML profiles. It defines Model, Package, and Subsystem, all of which serve as grouping units for other ModelElements. Models are used to capture different views of a software system. Packages are used within a Model to group ModelElements. A Subsystem represents a behavioral unit in a software system.

UML Profiles [9] (Sect. 4) are packages dedicated to group UML extensions: predefined sets of stereotypes, tagged values, constraints, and icons to support modeling in specific domains. Profiles are not formally defined in UML. This leaves us the space and flexibility to define our architectural profiles. The architecture model structure used in our approach is based on the UML model management package (see Figure 1).

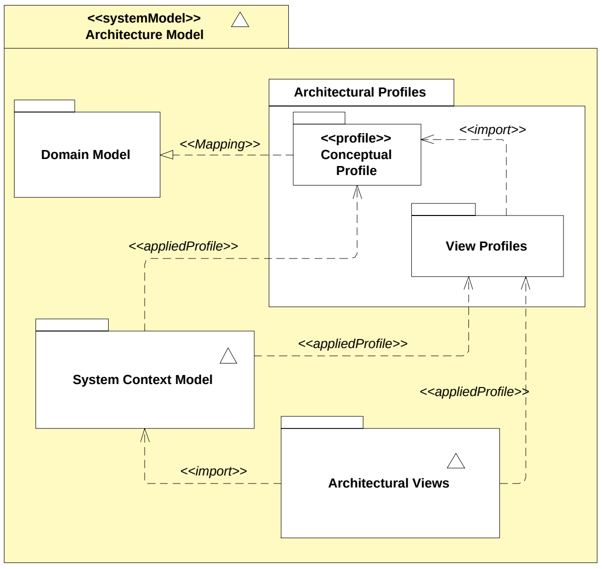

Figure 1: The structure of an architectural model

Two widely accepted approaches for working with multiple architectural views are the "4+1 view" model by Kruchten [6] and the "conceptual, module interconnection, execution, and code" view model by Soni et al. [12]. In practice we use a view model similar to the latter.

A *Domain Model* should be part of an architecture model; the term domain refers to the problem domain while all other parts of the architecture model belong to the solution domain. The Domain Model defines the key concepts and their relationships in the domain, and serves as a guideline for defining the corresponding UML extensions and conceptual models in the Architectural Profiles of the solution domain. The Domain Model in Figure 1 is included for model completeness; is out of the scope of this paper.

2.2 Architectural model structure

An architecture model structure includes three main parts: *Architectural Profiles*, *System Context Model*, and *Architectural Views*.

Architectural Profiles contain a *Conceptual Profile* and a set of *View Profiles*. All view profiles depend on the Conceptual Profile. The Conceptual Profile defines the fundamental concepts (types) including the concepts mapped from the domain model and the concepts introduced to implement these. These concepts are used throughout the entire architecture design and description. The Conceptual Profile should also contain a conceptual model that clearly specifies the architectural style and the validation rules with class diagrams and, if necessary, with OCL [9].

The architectural concepts specified in the Conceptual Profile define the nature of the system; they exist during the whole life cycle of a software architecture with different forms at different stages. Hence all architectural views should conform to the Conceptual Profile.

A view profile imports the Conceptual Profile and adds new concepts that are specific to this view. If the view profile has concepts that are in any form related with the concepts in the imported Conceptual Profile (i.e., specialization, containment, implementation, etc.), the relationship must be clearly described with UML associations. View profiles can be divided into two parts: a *stereotype definition part* and a *constraint definition part*. The former defines the stereotypes used in the architectural profiles and views, while the latter states the constraints that the views must conform to.

The stereotype definition part of a profile is given as a UML class diagram consisting of classes and dependencies between them. A class can either have stereotype ≪metaclass≫ or ≪stereotype≫. The name of the former class must be a standard UML metamodel element that is extended by the latter class. The name of the latter class declares a new user-defined stereotype. All dependencies can only be directed from a stereotype class into a metaclass class. Alternatively, the stereotype definitions can be given in a tabular form [9] (pp. 4-2 to 4-3).

The constraint definition part of a profile uses class and collaboration diagrams to specify the constraints, e.g., architectural styles or patterns, or the relationships allowed among certain types of elements. The classes used in constraint class diagrams are called *anonymous instance classes*. An anonymous instance class represents any instance class of the corresponding stereotype associated with it, i.e. it represents any instance class of a given stereotype. A name may be added after three dots "..." to increase the readability of the model.

Concrete software architecture is described with a System Context Model and several Architectural Views. The System Context Model defines the system border and deployment environment, typically showing how the system or subsystem connects to other (sub)systems through interfaces it implements or depends on.

Architectural Views package contains all the UML models describing the software system itself. A view should use only the subset of UML elements and extensions defined in the corresponding view profile(s), and follow the rules and constraints of the view profile. What views are needed in the system description may vary from case to case, depending on what main architectural concerns the architects are going to tackle.

A view can only be described with the subset of UML elements and extensions defined in the corresponding view profile(s) that it applies, following the rules of the corresponding profiles. This approach intentionally limits the verboseness of UML; we believe that this is necessary in order to achieve simpler, and thus clearer and more concise semantics for architectural modeling.

2.3 Architectural view and profile types

The architectural profiles and views are hierarchical. Figure 2 shows the dependency hierarchy of the architectural profiles and views used in this paper. In the following, we describe the architectural view types. The profiles are defined correspondingly.

A *structure view* (profile) shows the hierarchical decomposition of the system into subsystems and components, and describes the logical relations among them in the corresponding top-level view or sub-views. A structure view is defined using a class diagram.

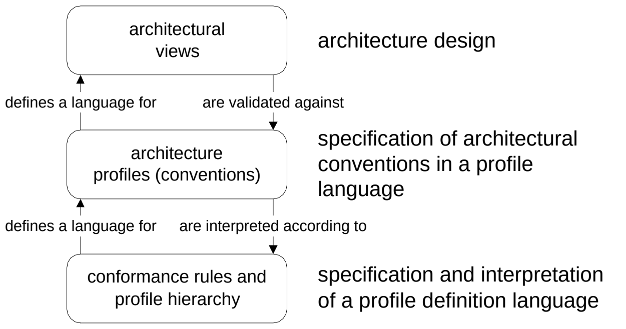
architectural views
architecture design
defines a language for
are validated against
architecture profiles (conventions)
specification of architectural conventions in a profile language
defines a language for
are interpreted according to
conformance rules and profile hierarchy
specification and interpretation of a profile definition language

Figure 3: Outline of an architecture validation process

A *behavior view* (profile) shows the realization of the main use cases as high-level interaction diagrams (i.e., sequence and collaboration diagrams) that are specified in the system requirements. On a more detailed level, a behavior view describes the realization of scenarios, unveiling the internal behaviors or activities of the system.

A *resource allocation* view (profile) shows the system at run-time. It shows the assignment of the components of the system to threads and processes. Stereotypes for threads, processes, and the communication means among the runtime entities, are defined in the corresponding view profile.

The other two view types are *build view* (profile) and *management view* (profile). A build view is a view of the system at build-time. The view should give the system builders a clear picture of what the entire system is built from, and if possible, a "makefile" for building the executables should be automatically generated from it. A management view is for the project managers to manage the development work. The view shows the responsible team or people for a subsystem or a component. Both build views and management views, together with their corresponding view profiles, are omitted in this paper, since they are not fundamentally relevant to the software architecture. There could also be other views concerning issues such as fault-tolerance and security.

3. VALIDATION RULES

3.1 Basic concepts

Figure 3 shows a general schema for arranging an architectural view validation process. Essentially, the profile hierarchy together with a configuration of *conformance rules* establishes a language for defining architectural profiles. A set of architectural profiles in turn describes the conventions and constraints for architecture design, i.e., it defines a language for defining architectural views.

The rules can be divided into three categories: (1) profile conformance rules, (2) view consistency rules, where views must be mutually consistent, and (3) profile consistency rules, where profiles must be mutually consistent. These categories are shown in Figure 4.

One simple example of a model consistency rule is to require that operations called in a behavior view must be defined in a structure view, either by the base class of the instance whose operation is called, or by an interface the class implements. Another example would be that every link and its end instances in a behavior view must have a

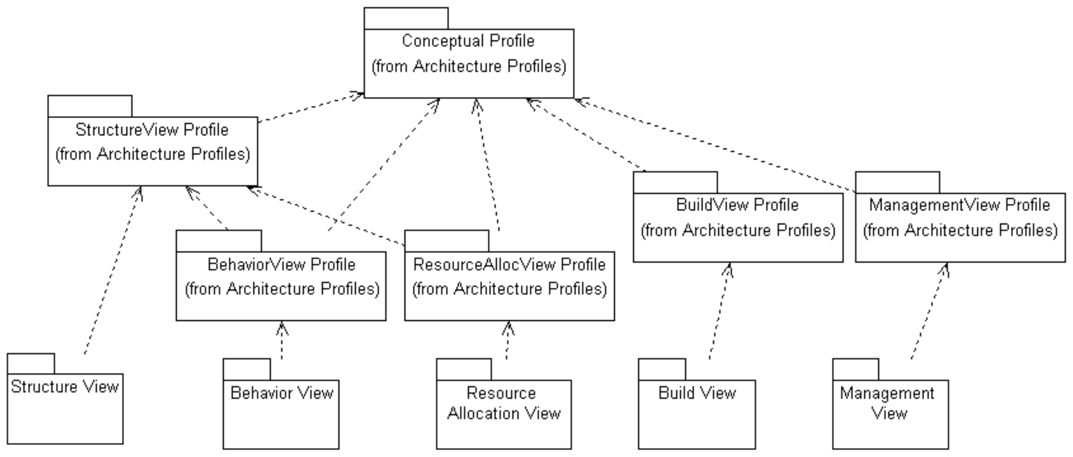
Conceptual Profile
(from Architecture Profiles)
StructureView Profile
(from Architecture Profiles)
BehaviorView Profile
(from Architecture Profiles)
ResourceAllocView Profile
(from Architecture Profiles)
BuildView Profile
(from Architecture Profiles)
ManagementView Profile
(from Architecture Profiles)
Structure View
Behavior View
Resource Allocation View
Build View
Management View

Figure 2: Dependencies between architectural views and profiles

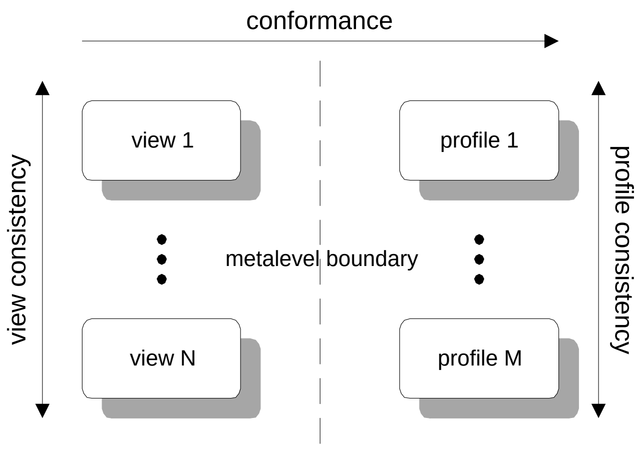

Figure 4: Rule categories

base association and base classifiers, respectively. While important, this paper omits further discussion on consistency and focuses on profile conformance rules. For discussion on UML and model consistency, see e.g. [7].

Since architectural profiles and views reside at different metamodeling levels, we must define when a model element in a view is an instance of an element defined in a profile. This inter-model *correspondence relationship* is defined for the most important elements as follows:

1. Two classifiers correspond to each other if their stereotypes are the same.

2. Two relationships (i.e., dependency, association) correspond to each other, if their stereotypes are the same and there exists a bijective mapping between all the participating classifiers from one relationship to another, such that these classifiers correspond to each other.

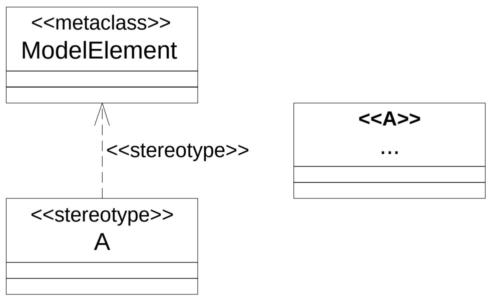

Figure 5: Stereotype conformance

3. Two instances correspond to each other if their stereotypes are the same and their base classifiers correspond to each other.

4. Two links correspond to each other if their stereotypes are the same and there exists a bijective mapping between all the participating instances from one link to another, such that these instances correspond to each other.

We will refer to these definitions when defining the actual conformance rules in Subsection 3.2.

3.2 Conformance rules

A conformance rule defines how a particular structure in a profile is interpreted, dictating how a view is validated against it. A conformance rule is evaluated between a view and a profile. We present five examples of profile conformance rules that can be applied between profiles and views.

3.2.1 Stereotype conformance

Definition 1. Every stereotype in a view must be defined in a stereotype definition profile belonging to the profile hierarchy. Every classifier in a view must have a stereotype.

The left-hand side of Figure 5 shows a stereotype definition profile with a standard UML metaclass ModelElement

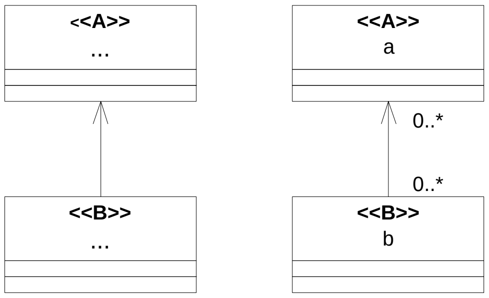

Figure 6: Relationship conformance

and a new, user-defined stereotype, ≪A≫. The right-hand side shows a class belonging to an architectural view whose stereotype conforms to the profile. This rule is applied to both profiles and views; with profiles, the rule declares consistency.

The ModelElement metaclass is defined abstract by UML and thus stereotypes based on it cannot themselves appear in architectural views. Typical — and more useful — base classes for user-defined stereotypes include Package, Subsystem, Class, Interface, and Association.

3.2.2 Relationship conformance

Definition 2. Every relationship (i.e., association, dependency) in a view is required to have a corresponding relationship in a profile belonging to the profile hierarchy.

The left-hand side of Figure 6 shows a constraint profile that allows an association between elements having stereotypes ≪A≫ and ≪B≫. The right-hand side shows a conforming architectural view with a corresponding association between classes ≪A≫ a and ≪B≫ b.

3.2.3 Multiplicity conformance

Definition 3. The number of associations each classifier in a view participates in must fall in range of the multiplicities defined by the corresponding association in a profile belonging to the profile hierarchy.

The right-hand side of Figure 7 shows two associations leaving from class ≪B≫ b, one to ≪A≫ a1 and the other to ≪A≫ a2. The number of these associations conform to the multiplicity ranges defined by the corresponding association on the left-hand side constraint profile.

Multiplicity conformance also implies the existence of classifiers. If there exists a classifier in a view, and a corresponding classifier in a profile participates in an association having all lower multiplicity bounds of the other ends greater than zero, a corresponding association and participating classes must exist in a view belonging to the view hierarchy. The structure in the left-hand side of Figure 7 generates such a constraint: the existence of a classifier with stereotype ≪B≫ requires the existence of two assocations and their end elements with stereotypes ≪A≫, but not vice versa.

3.2.4 Interface conformance

Definition 4. For every interface in a view, if there exist a corresponding interface and a (set of) realizing class(es) in the profile hierarchy, some corresponding realizing class must also exist in a view belonging to the view hierarchy.

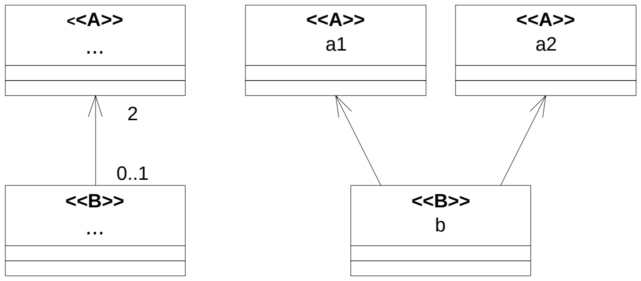

Figure 7: Multiplicity conformance

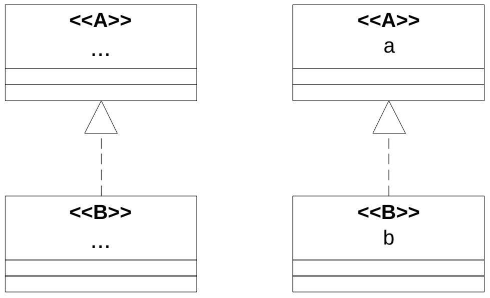

Figure 8: Interface conformance

The left-hand side of Figure 8 shows a constraint profile allowing a realization relationship between classes having stereotype ≪B≫ and classes having stereotype ≪A≫, and at the same time requires an instance of ≪A≫ to have (at least one) realizer. The right-hand side shows a conforming architecture view with a corresponding realization relationship between ≪B≫ b and ≪A≫ a.

3.2.5 Link conformance

Definition 5. Every link in a (behavior) view must have a corresponding link in a (behavior) view profile belonging to the profile hierarchy.

The left-hand side of Figure 9 shows a behavior view profile that defines a link between instances of classes ≪A≫ and ≪B≫. The right-hand side shows a behavior view with a corresponding link.

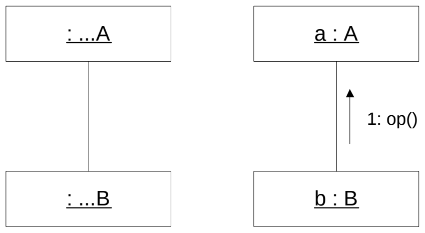

Figure 9: Link conformance

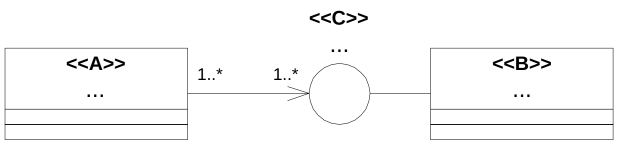

Figure 10: Relationship indirection

3.3 Auxiliary definitions

In addition to the conformance rules, we give two auxiliary definitions. These definitions are not rules by themselves, but rather "preprocessing" constructs for generating additional, or removing existing, constraints on model elements before starting the actual validation scheme.

3.3.1 Relationship indirection

Definition 6. Every relationship (i.e., dependency, association) can exist either directly or via an interface. For every classifier depending on an interface, and every classifier realizing this interface, there exists an implicit corresponding relationship between the using classifier (client) and the realizing classifier (supplier) with identical characteristics (e.g. multiplicities).

This definition is used by the other rules for generating indirect dependencies not explicitly defined in a profile. The definition is applied to profiles in the profile hierarchy. Figure 10 shows a constraint profile defining how an ≪A≫ class uses a ≪B≫ class through a ≪C≫ interface. By dependency indirection, there exists an implicit association between the ≪A≫ class and the ≪B≫ class with the same multiplicity range (1..*, 1..*). One typical usage scenario for this operation is to allow the existence of direct component to component dependencies for generating more abstract diagrams.

3.3.2 Stereotype generalization

Definition 7. Each element inherited from a user-defined stereotype is subject to, or inherits, the constraints placed upon its parent in the profile hierarchy.

This operation extends the definition of how a stereotype definition part of a profile can be constructed by allowing the user to define new subconcepts based on higher-level user-defined stereotypes. This operation is useful when e.g. the user wants to define common constraints for a set of structurally equivalent, but semantically separate, modeling concepts. Figure 11 shows an example of stereotype generalization definition, where a subconcept represented by stereotype ≪B≫ is derived from concept represented by stereotype ≪A≫. As a result, all constraints placed on elements having stereotype ≪A≫ are inherited by elements having stereotype ≪B≫.

4. EXAMPLE: AN IMAGE RECOGNITION SYSTEM

This section gives an example of validating architectural views against a set of profiles. Figure 12 shows a deployment view of the example system. The system runs in Application

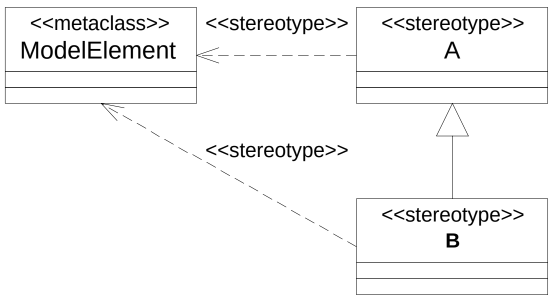

Figure 11: Stereotype generalization

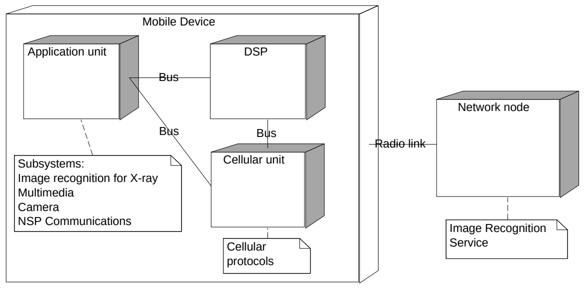

Figure 12: Deployment diagram of example system

unit node of Mobile device and is connected to DSP and Cellular unit nodes via two buses. The Mobile device is further connected to Network node via Radio link. The example is modified based on a current case study on one of Nokia's product platforms. It is simplified and generalized for the purposes of this paper, but still complex enough to demonstrate our approach. Figure 14 shows the context model of the example system. The system, "Image recognition for X-ray", is an application developed for mobile terminals with an attached camera.

The system implements an image recognition service: after a picture is taken using the integrated camera component, it is converted into a suitable format and sent over to a network node that tries to recognize the image. The result is propagated back to the user. For space limitations, the stereotypes defined in the conceptual profile and structure profile are given in tabular form in Table 1. The table shows the base classes and the new stereotypes based on them, as defined in conceptual and structural profiles.

The conceptual profile for the example system is given in Figure 13. The profile defines higher-level architectural entities and their inter-relationships in the architecture design. Figure 15 shows a structure view of the architecture. Architecture violations are marked with numbers from one to four:

1. The stereotype used by ≪Session≫ "Socket server IF" is not defined in the stereotype definition profiles, thus violating the stereotype conformance rule.

2. As a consequence of item 1, the dependency between ≪Server≫ "ImageRecogServer" and ≪Session≫ "Socket server IF" is invalid and violates the relationship conformance rule.

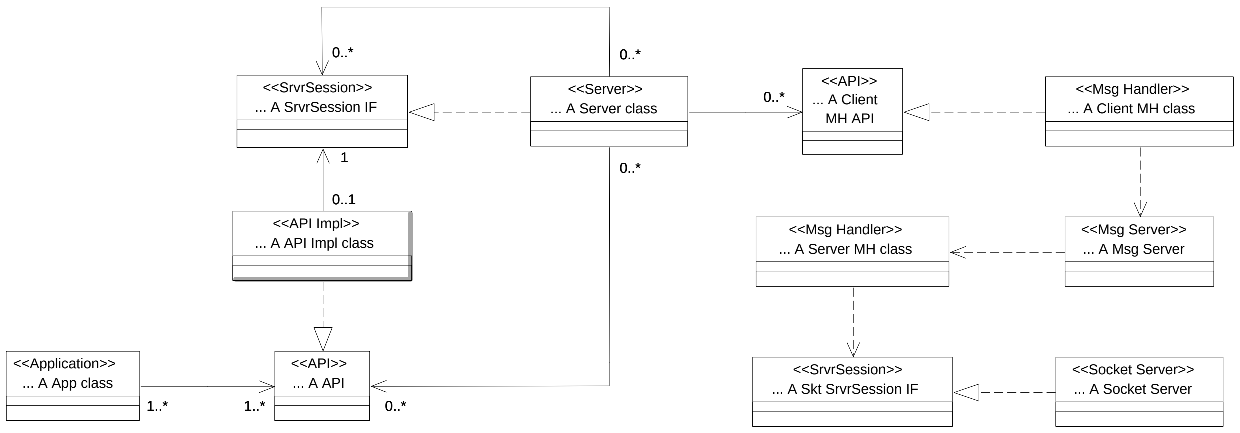

Figure 13: An example Conceptual Profile

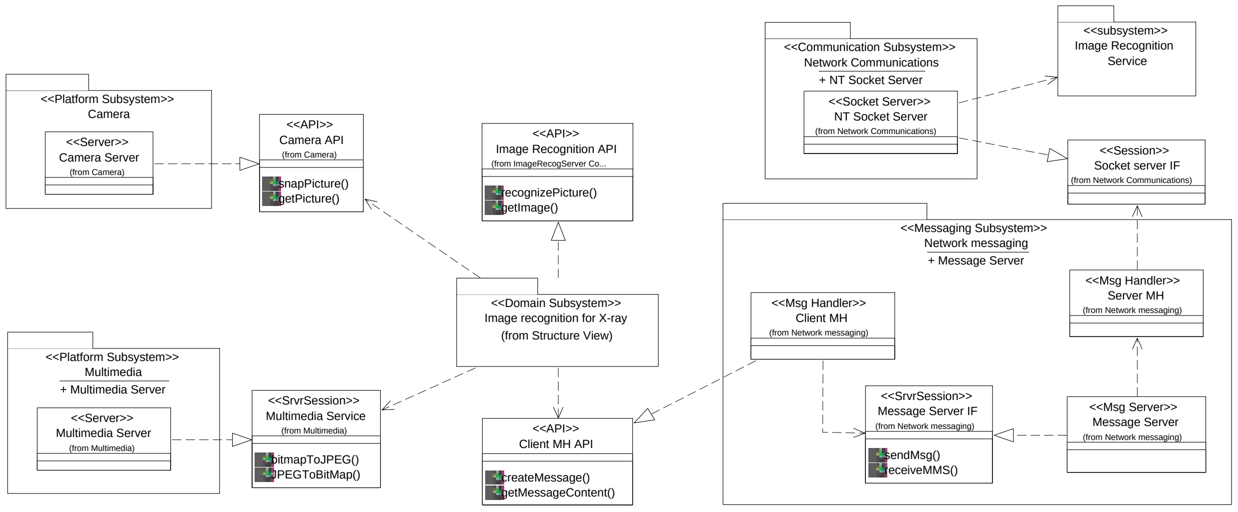

Figure 14: An example System Context Model

Table 1: Example stereotype definitions

Conceptual profile		Structure profile	
Base Class	**Stereotype**	**Base Class**	**Stereotype**
Class	≪Application≫ ≪Msg Server≫ ≪Socket Server≫ ≪Server≫	Subsystem	≪Domain Subsystem≫ ≪Platform Subsystem≫ ≪Messaging Subsystem≫ ≪Communication Subsystem≫
Utility	≪Msg Handler≫ ≪API Impl≫	Package	≪App Component≫ ≪Server Component≫
Interface	≪API≫ ≪SrvrSession≫	Association	≪Message≫ ≪FunctionCall≫

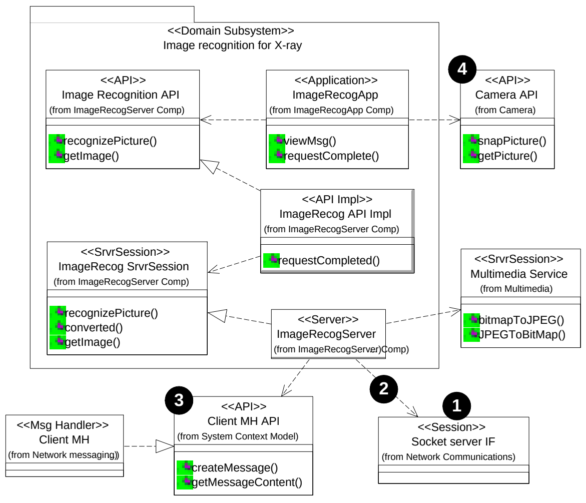

Figure 15: An example structure view

3. By the multiplicity conformance rule, interface ≪API≫ "Client MH API" requires an association to a class having stereotype ≪Application≫. This association is not present and consequently violates the rule.

4. By the interface conformance rule, interface ≪API ≫ "Camera API" should have a realizing class. The realization in Figure 14 between ≪Server≫ "Camera Server" and ≪API≫ "Camera API" is invalid and therefore violates the rule.

Otherwise the structure view conforms to the conceptual profile: the remaining relationships presented in Figure 15 are conforming and thus legal dependencies and realization relationships between classes, utilities, and interfaces.

5. IMPLEMENTATION

In order to evaluate the techniques described in this paper, a tool, *artDECO*, implementing the validation rules was developed. The operations are implemented on top of a tool-independent UML processing platform, xUMLi [1]. The platform enables users to build various kinds of UML model processing facilities as individual components, and combine and use them from integrated CASE tools. The platform conforms to the UML metamodel version 1.4 and is currently integrated with Rational Rose.

The profile hierarchies are stored as package hierarchies in a local Rose repository. The architecture checking tool uses xUMLi to import the necessary UML models from Rose. Each architecture model is checked against the validation rules, implemented as xUMLi components using the platform's OCL interpreter. The results are presented to the user in a simple graphical dialog and in a complementary textual format (in XML). The user can browse the architectural mismatches resulting from different operation categories. As the user selects a particular mismatch, the diagram containing the corresponding element is opened in Rose and the element itself is selected. This allows the user to quickly discover and browse through the non-conformant structures and correct them if necessary.

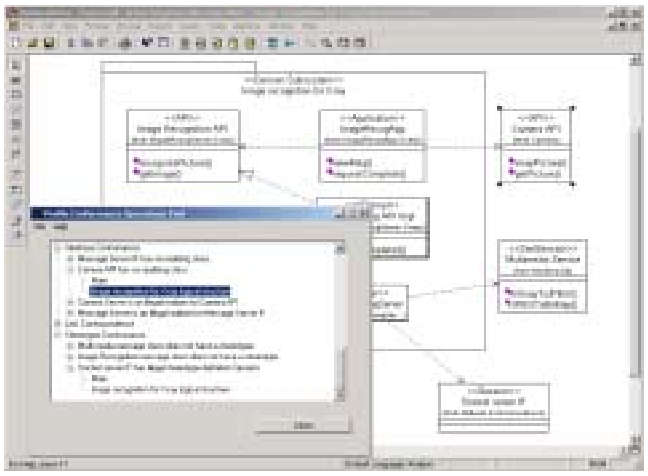

Figure 16: Screenshot of the artDECO tool

Figure 16 shows a screenshot of the artDECO tool running on top of Rational Rose. The results from applying the rules are shown in a dialog, and the corresponding window is shown together with the element selected.

6. CASE STUDY

The toolkit and our approach have been evaluated using a system very similar to the one described in Section 4. This case study focused on designing the architecture of "X-ray image application system" on one of Nokia's product platforms. Most architectural profiles used in this case study are common to all application systems on the same platform. The set of common profiles is configurable to emphasize different requirements. Product-specific architectural profiles can also be added to validate the architecture design of a certain product.

The case study consisted of two subsystems and had connections to five other subsystems. The set of profiles contained 11 diagrams and 78 classifiers, while the set of views contained 14 diagrams and 101 classifiers. The model was constructed by a person who was familiar with the architectural conventions of the platform, imitating the work of a typical designer.

The tool was able to find 39 architectural mismatches in the case study. Table 2 lists the rules, the mismatch types, and the number of mismatches found in this case study. While the quantitative data merely displays anecdotal evidence, it still goes on to show that even a system of a relatively small size can contain a considerable amount of profile violations.

Not surprisingly, an overwhelming number of mismatches involve relationships. These mismatches are expected to emerge in situations where the architects want to connect elements without paying enough attention to the architectural styles and constraints imposed by the profiles (e.g. interface-based decoupling of entities).

The evaluated system followed the Nokia guidelines for architecture design and was based on existing product line architecture models. Consequently, the system had real-life characteristics. The profile hierarchy and the set of confor-

Table 2: Architectural mismatches found

Rule	Mismatch type	No
Stereotype conformance	Missing stereotype	2
Relationship conformance	Illegal dependency	12
	Illegal association	5
Multiplicity conformance	Multiplicity out of range	5
Interface conformance	No realizing class	1
	Illegal realizing class	4
Link conformance	Illegal link	6

mance rules were specifically designed to meet the requirements of the target system architecture provided by Nokia. Still, we argue that the given conformance rule configuration is useful also with a general software development process. The underlying generic principles and the technical solutions of the tool make it possible to reconfigure artDECO for an alternative architectural process.

Even though the system itself was relatively small, it became evident that validating the architectural views using the rules by hand was much too laborious. To give an estimate, according to our experience it takes an average of two hours to check the system by hand for an individual familiar with the rules, profiles, and architectural views; still, a significant portion of the mismatches will remain undetected. Suitable tool support seems to be a prerequisite for systematically enforcing the validity of the system under design.

We believe there hardly exist any universal architectural profiles that could be used to validate arbitrary software architecture. The profiles should be specific to product lines or even single products in order to produce practical and valuable architecture model validation results. However, our toolkit and approach to profile construction and model validation are general enough to be applied in different domains and different product lines. The set of profiles is expected to stay relatively stable even if the checking process is performed for a larger architecture model.

7. RELATED WORK

While there exists relatively small amount of research on using the UML profile mechanism for architectural validation, a considerable effort has been placed on defining relations between ADLs and UML, focusing on mapping the concepts of the former into the visual notation of the latter. This is usually done by either using the standard extension mechanisms or by altering the UML metamodel itself, the main motivation being the possibility of using the existing UML CASE tools with ADLs.

Zarras et al. [13] define the concepts of a general ADL using a UML profile. After mapping the primary ADL concepts (component, connector, configuration) into UML notational elements, a base profile is defined that can be further extended to match specific architectural styles or views. Another example is presented by Hudaib and Montangero [4], who use UML profiles to map the core concepts of an architectural specification language, based on temporal logic, to UML. This allows the usage of formal specification and analysis tools on UML models. An example of an approach that modifies the UML metamodel is presented by Kandé and Strohmeier [5].

Egyed [2] discusses mapping ADL specifications into UML models using a view integration process. He also defines conformance and consistency relationships, but between UML models, not between profiles and views. He aims at mapping and integrating ADL models into a UML system model by defining mapping, transformation, and differentiation procedures.

The main difference of our approach is that we are not defining a single profile. Instead, with a profile hierarchy and a set of validation rules, we establish a language for defining profiles. Our approach focuses on giving support for configurable validation processes that can be customized to support the constraints and conventions of a given product line or domain. In addition, our approach relies solely on UML.

8. CONCLUDING REMARKS

Based on our initial experiences, the techniques described in this paper were found promising. Tool-supported validation of architectural design can significantly reduce the number of architectural mismatches, thus avoiding design errors that would deteriorate the architecture and possibly lead to costly re-engineering tasks in the later development phases.

For the time being, the rules are structural in nature. As such, the rules are nevertheless significant in practice. Issues like behavioral patterns are beyond the approach described here. While the rules originally stem from pragmatic needs, the approach is already quite general, configurable, and extensible. Topics for our future research include, for example, constructs not used so far by the rules (e.g. general OCL constraints, meta-attributes), and the use of the UML diagrams types not addressed in this paper. It is also our goal to further explore, classify, and categorize the concepts integral to the profile-centric approach.

During year 2003, we will perform two large-scale industry case studies in collaboration with Nokia's business units to further evaluate our approach. Our targets are two main product lines of Nokia Mobile Phones, at the level of both a product line, and a specific product. For this purpose, we are aiming at integrating our techniques with software development processes at Nokia.

The next version of the validation tool will support freely configurable architectural validation, allowing the architects to define a relevant set of conformance rules and a model hierarchy for a process of choice. The goal of our work is to develop a general-purpose, profile-based architecture modeling and validation approach with adequate tool support.

9. ACKNOWLEDGEMENTS

The authors wish to thank Jani Airaksinen for the implementation of the artDECO tool, Kai Koskimies and Tarja Systä for their valuable comments, Antti-Pekka Tuovinen for the initial case study material and view definitions, and the PRACTISE/UML team at Tampere University of Technology and the ART project team at Nokia Research Center. This work has been financially supported by the Academy of Finland (UML++ project).

10. REFERENCES

[1] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, and T. Systä. xUMLi: Torwards a Tool-independent UML Processing Platform. In K. Osterbye, editor, *Proceedings of the Nordic Workshop on Software Development Tools and Techniques, 10th NWPER Workshop*, pages 1–15. Copenhagen, Denmark, IT University of Copenhagen, August 2002.

[2] A. Egyed and N. Medvidovic. Extending Architectural Representation in UML with View Integration. In R. France and B. Rumpe, editors, *Proceedings of the Second International Conference on the Unified Modeling Language, UML'99*, pages 2–16. Fort Collins, CO, USA, Springer, October 1999.

[3] C. Hofmeister, R. Nord, and D. Soni. Describing Software Architecture with UML. In P. Donohoe, editor, *Proceedings of Working IFIP Conference on Software Architecture*, pages 145–160. San Antonio, Texas, USA, Kluwer Academic Publishers, February 1999.

[4] A. Hudaib and C. Montagero. A UML Profile to Support the Formal Presentation of Software Architecture. In *Proceedings of 26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment*, pages 217–223. Oxford, England, IEEE Computer Society, August 2002.

[5] M. M. Kandé and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference*, volume 1939 of *Lecture Notes in Computer Science*, pages 513–527. York, UK, Springer, 2000.

[6] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 28(11):42–50, November 1995.

[7] L. Kuzniarz, G. Reggio, J. Sorrouille, and Z. Huzar. Proceedings of the Workshop on Consistency Problems in UML-based Software Development. Blekinge Instutute of Technology Research Report 2002:06, 2002.

[8] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, January 2002.

[9] OMG. Omg unified modeling language specification, version 1.4, september, 2001. On-line at http://www.omg.org.

[10] Rational Software Corporation. Rose Enterprise Edition, 2003. On-line at http://www.rational.com/products/rose.

[11] C. Riva, J. Xu, and A. Maccari. Architecting and Reverse Architecting in UML. In A. Brown, W. Kozaczynski, P. Kruchten, and G. Larsen, editors, *Proceedings of ICSE 2001 Workshop for Describing Software Architecture with UML*, pages 88–93. Toronto, Ontario, Canada, IEEE Computer Society, May 2001.

[12] D. Soni, R. L. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In *Proceedings of International Conference on Software Engineering ICSE 1995*, pages 196–207. Seattle, Washington, USA, April 1995.

[13] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Kguyen. Towards a Base UML Profile for Architecture Description. In A. Brown, W. Kozaczynski, P. Kruchten, and G. Larsen, editors, *Proceedings of ICSE 2001 Workshop for Describing Software Architecture with UML*, pages 22–26. Toronto, Ontario, Canada, IEEE Computer Society, May 2001.