Kai Koskimies, Ludwik Kuzniarz,
Jyrki Nummenmaa and Zheying Zhang (eds.)

Proceedings of the NWUML'2005: The 3rd Nordic Workshop on UML and Software Modeling

TIETOJENKÄSITTELYTIETEIDEN LAITOS
TAMPEREEN YLIOPISTO

A-2005-3

TAMPERE 2005

Preface

Welcome to NWUML'2005: The 3rd Nordic Workshop on UML and Software Modeling, which is held in Tampere, Finland, on 29.-31.8.2005.

The intention of NWUML is to bring together researchers and Ph.D. students in the fields of software modeling, including model-based development methods and tools in general and UML-based modeling in particular. NWUML intends to be an open forum with ample time set aside for panels and discussions. The workshop intends to function as a platform for establishing co-operative research projects between the participants in the region.

Even though there are other events on this area worldwide, we in particular hope that a Nordic research community is formed and strengthened. The actual workshop program is designed to be such that it leaves place for discussions and thereby hopefully supports future co-operation between participants of the workshop.

The past workshops in this series were organized in Ronneby, Sweden (2003) and Turku, Finland (2004).

We hope for an interesting and fruitful workshop.

Editors
Tampere, August 2005

Program Committee

Anders Ek, Telelogic AB, Sweden
Juha Gustafsson, University of Helsinki, Finland
Klaus Marius Hansen, University of Aarhus, Denmark
Sune Jakobsson, Telenor, Norway
Kai Koskimies, Tampere University of Technology, Finland (Chair)
Ludwik Kuzniarz, Blekinge University of Technology, Sweden
Johan Lilius, Åbo Akademy University, Finland
Jyrki Nummenmaa,University of Tampere, Finland
Ian Oliver, Nokia Research Center, Finland
Ivan Porres, Åbo Akademy University, Finland
Andreas Prinz, Agder University College, Norway
Miroslaw Staron, Blekinge University of Technology, Sweden

Organization Committee

Kai Koskimies, Tampere University of Technology, Finland
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden
Jyrki Nummenmaa, University of Tampere, Finland
Ivan Porres, Åbo Akademi University, Finland
Andreas Prinz, Agder University College, Norway
Zheying Zhang, University of Tampere, Finland

Table of Contents

Guidelines for creating “good” stereotypes

Miroslaw Staron, Ludwik Kuzniarz

Department of Systems and Software Engineering, School of Engineering
Blekinge Institute of Technology, Ronneby, Sweden
(miroslaw.staron, ludwik.kuzniarz)@bth.se

Abstract. Stereotypes are a means of virtually extending the set of modeling constructs available in a modeling language like the Unified Modeling Language (UML). The use of stereotypes depends on the purposes which the stereotypes are aimed at. In this paper we present a set of guidelines for creating stereotypes in an effective way for specific purposes. The guidelines are aimed at creating “good” stereotypes – i.e. stereotypes which are fit for the purpose they should serve and they are appropriately specified. In our studies we identified eight distinct groups of stereotypes for which the guidelines are defined. The guidelines are elaborated based on investigation of several existing UML stereotypes in a set of empirical studies.

1. Introduction

Using precise models and precise modeling languages to create the models underpins the effectiveness of model-based software development (a.k.a. Model Driven Architecture, [1]). General-purpose modeling languages, however, rarely provide purpose specific constructs to enable effective usage of models in an automated way. The Unified Modeling Language (UML, [2, 3]) provides means for creating precise models by utilizing extension mechanisms inherent in the language thus customizing the language. Creating suitable language extensions, nevertheless, requires extensive knowledge of both the domain for which the customization is done and the means of creation of suitable extensions. The main extension mechanism in UML is the notion of *stereotype* which is a means of branding existing modeling elements with additional properties and semantics thus extending the “vocabulary” accessible for modelers. Stereotypes are grouped into *profiles* which are sets of stereotypes aimed to serve one purpose (e.g. UML Profile for CORBA [4] which is aimed at providing constructs enabling precise modeling of CORBA based software). The stereotypes are dedicated for modelers who “need a richer vocabulary to describe their design intentions” [5]. The modelers need to automate their software development process (e.g. [6]), increase comprehension of their models (e.g. [7, 8]), or increase benefits from using models in software development by maintaining certain attributes of software constant (e.g. [9]) thus facilitate domain specific modeling within a framework of a single (general-purpose) modeling language. Guidelines presented in this paper provide aid in creating extensions to UML using stereotypes that would improve the practice of modeling. Based on the needs of particular enterprises, the guidelines allow choosing between very light (shallow) customizations (i.e. by creating simple stereotypes in a very informal way) and more powerful (deep)

customizations (i.e. by creating stereotypes that introduce significant changes to the extended modeling element). The guidelines allow also creating stereotypes which are "good" – i.e. suitable for the purpose they are supposed to serve. The guidelines contain instructions on which elements should be in the stereotype definition. For example how much documentation should the stereotype have so that it contains enough information (c.f. **Extensive documentation**, [6]) and, at the same time, is concise (c.f. **Concise profiles** and **Simple profile definition**).

In our previous study on a realization of a vision of model driven software development (Model Driven Architecture, MDA [1]) in industry we have investigated the way in which language extensions – profiles and stereotypes – are used in practice to make the modeling language precise enough to enable automating software development with transformations. The results of the case study showed that language customization is the basis for realizing the vision of model driven software development. The guidelines presented in this study are aimed at aiding enterprises willing to effectively customize their modeling language. In addition to the factors related to documentation (mentioned in the previous paragraph) the guidelines address these factors: (i) **Improvement of the current way of working** - stereotypes allow introducing extensions to the base modeling language without the need for its complete redefinition; (ii) **Integration into currently used tools** - stereotypes can be used within UML modeling tools without the need for introducing new tools.

We have investigated several UML profiles used in industry. We have, however, not limited ourselves to only the standardized profiles. A set of non-standardized profiles used in specific companies for very narrow purposes was also investigated in order to capture the state-of-practice in using stereotypes in particular companies for their in-house language customizations. The identified purposes reflect the empirical research conducted on several stereotypes from different "vendors" over four years.

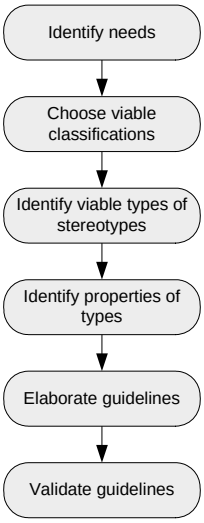


Fig. 1. Process of elaborating the guidelines

The process of elaborating the guidelines is presented in Fig. 1. The first step in elaborating the guidelines was the identification of the needs which was done based on the industrial case study on customizations of UML in [10]. The identified needs stem from the factors presented in the introduction to this paper.

The second step was to choose the appropriate method of elaborating the guidelines. We have chosen an empirical approach which was based on investigation of several existing UML profiles and on existing classifications of stereotypes which reflect the usage of stereotypes in practice. In order to adjust our method and decrease subjectivity in our study we have performed an auxiliary controlled experiment with several subjects which aimed at verification whether our approach to choosing viable classifications is appropriate. The initial set of three classifications has been afterwards refined to contain only two classifications as the third one was found to classify all stereotypes in UML into one category.

After choosing the viable classifications we have classified a set of 98 stereotypes in order to find dependencies between categories in different classifications. The viable types of stereotype have been examined in the next step in order to identify common properties of stereotypes of these types. The common properties can be used as a set of quality assessment criteria for stereotypes (to check whether newly created stereotypes are appropriate and good). This part of our research is presented in [11]. The common properties of stereotypes are inputs for guidelines on how to create "good" stereotypes which are presented in this paper.

2. Related work

The basic guidelines on how to document stereotypes can be found already in the UML specification. Due to the fact that the specification was created for general usage, the guidelines are basic ones and should be refined. An analysis of the way in which stereotypes can be created by users in UML can be found in [12]. This analysis is the basis for consideration on which elements should be used to properly define stereotypes. A set of good practices from [13-17] was used as a basis for elaborating the guidelines on how stereotypes should be used with respect to the appropriate layer in the four layer metamodeling framework [18]. We have used the analysis of stereotypes and profiles presented in [19] to explicitly choose one placements of stereotypes in the four-layer metamodeling framework – i.e. that stereotypes should be placed at the metamodel layer since they define entities that are to be instantiated (not inherited from) in the model layer where the stereotypes are to be used.

Usage scenarios identified in [17] provided us with a better overview of what are alternative perceptions on the usage of stereotypes which we incorporated in the research leading to elaborating of these guidelines.

While elaborating guidelines presented in this paper we investigated the existing guidelines on how to use stereotypes in general – e.g. [20, 21].

3. Guidelines for creating good stereotypes

The results of the studies of existing stereotypes which led to elaboration of the guidelines showed that all stereotypes should possess properties of *type classification* stereotypes as identified by Atkinson et. al. [17]. The stereotypes should create virtual kinds of modeling elements (i.e. metaclasses) to be instantiated in models. This means that semantics of the stereotypes should relate to the semantics of the *kind* of modeling element (i.e. the base element), hence the first guideline that applies to all stereotypes: *stereotypes should add properties to elements in UML and not to their instances*. If a stereotype adds properties to instances of elements (not to definitions of elements) it should be replaced by inheritance.

The process of creating and using stereotypes which is supported by the guidelines presented in this paper is as follows:

1. Find which role should be played by the stereotype in designs (using the decision graph presented in Fig. 2).

2. Find which properties the stereotype should have (using the decision graphs in appropriate sub-sections).

The first step is to find the appropriate role. This is done using the decision diagram in Fig. 2 by examining the intended purpose of the stereotype. This should be done by the person responsible for creating the stereotype.

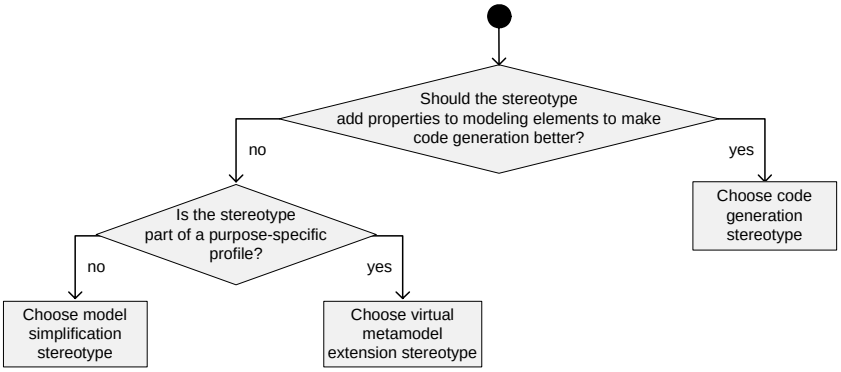


Fig. 2. Choosing the purpose of the created stereotype

The first decision to be taken is whether the stereotype should be used for code generation. The reason for distinguishing this role is that stereotypes for code generation were found to be specially designed and used in modeling. These are described in further in this paper. If the stereotypes are not designed for code generation, then an important aspect is whether they are defined for a *purpose-specific profile*. If they are a part of such a profile then they should be created in a more detailed way. If a stereotype is not part of a purpose specific profile, then it should be created in a less strict way depending on the further purpose of the stereotype. This is further explained while discussing guidelines in this paper.

Choosing the appropriate purpose of the stereotype influences the way in which the stereotypes should be defined, documented and used. Within each purpose of the stereotype there are stereotypes that introduce various degrees of modifications to their base elements.

3.1. Model simplification stereotypes

Once the model simplification stereotype is chosen, it should be further investigated what is the intention of the stereotype. This investigation is presented in the following graph by taking the decision D1:

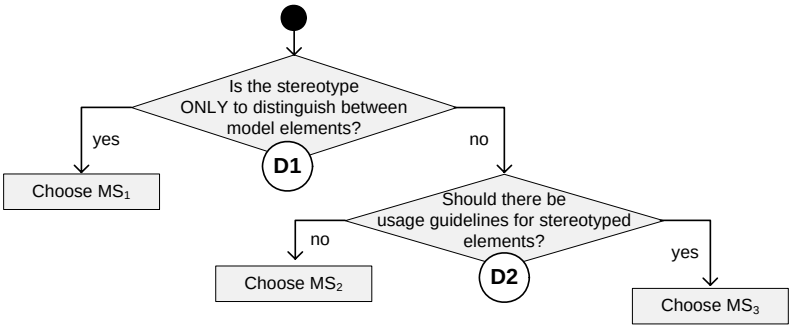


Fig. 3. Choosing between the kind of model simplification stereotypes

If the stereotype is intended to be used by a small group of people (e.g. within a single project) and thus the stereotype is mainly to help distinguish between particular elements in the design then the stereotype (MS_1):

- Can be created
  - on-demand basis during the project.
  - by individual developers.
- Should not contain tag definitions.
- Should not contain constraints.
- Should have documentation that includes:
  - Name and representation (icon if required) of the stereotype
  - Description of the stereotype (i.e. intention and meaning of the stereotype)
- Can contain icons if the purpose of the stereotype is to increase comprehension of models (the icons itself should reveal the intention of the stereotype – i.e. be intuitional).
- Can extend both an abstract and a concrete metaclass.
- Should not introduce new semantics to the stereotyped model element.

The documentation of these stereotypes can be rather informal and it can be a note containing the description of the intention and the meaning of the stereotype. The note should be attached to the definition of the particular stereotype.

These stereotypes can even be created for individual needs by modelers who want to emphasize certain aspects of their designs in order to improve their personal productivity. An example of a stereotype defined by an individual modeler to designate elements which the modeler finds viable to be later stored persistently is defined in Fig. 4.

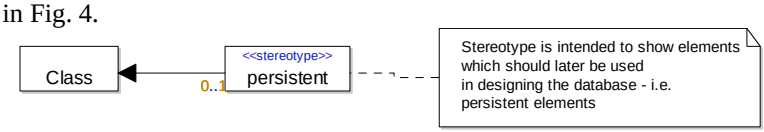


Fig. 4. Example of MS_1 – stereotype «persistent»

If the stereotype should be more widely spread or should be used for more than a mere distinguishing between particular elements, then it should be decided (decision D2) whether the stereotyped elements should be used in specific situations – i.e. have specific usage guidelines. If it should not, then the stereotype (MS_2):

- Can be created
  - on-demand basis, and
  - by modelers who identified the need for the stereotype.
- Can contain icons if the stereotype should also be used to increase the comprehension of created models.
- Should not contain tag definitions nor constraints.
- Should include documentation (can be just a note) which consists of:
  - Name and representation of the stereotype
  - Description of the stereotype (i.e. intention and meaning of the stereotype)
  - Description of the intention and meaning of the defined tag definitions
  - An example of how tag definitions of this stereotype should be used.
- Should not introduce new semantics to the stereotyped elements.

An example of this kind of stereotype is a more advanced version of the «persistent» stereotype which is aimed at providing also information if the stereotyped element should be stored in a database.

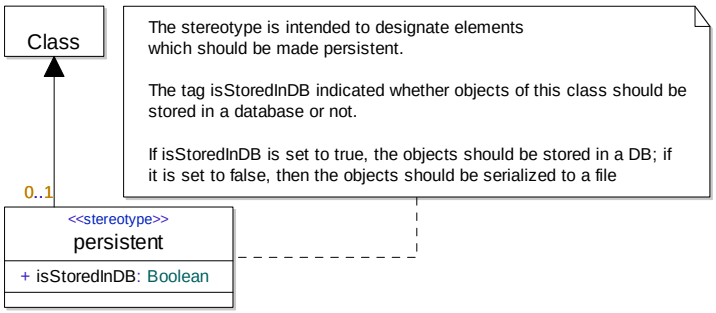


Fig. 5. Example of MS_2 stereotype - «persistent»

If the stereotyped elements should be used in a specific way then it should provide usage guidelines for the stereotyped elements (decision D2). Stereotypes which are intended to provide guidelines (MS_3):

- Should be created
  - as part of a software process improvement activity, and
  - by modelers who work with software process improvement issues and are aware of the implications of different issues related to stereotyping.
- Can contain icons, but the icons should not be made the main part of stereotype definition.
- Can contain tag definitions which add new properties to the extended model element.
- Should contain constraints restricting usage of the stereotyped elements – i.e. constraints stating when the stereotyped element cannot be used.
- Be documented in a note or a separate document which contains:
  - Name and representation of the stereotype
  - Description of the stereotype (i.e. intention and meaning of the stereotype)
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.

- Description of the intention, meaning (expressed in natural language) of the constraints defined for the stereotype.
- An example of using the stereotyped elements

An example of an MS_3 stereotype is even more advanced version of the stereotype «persistent» which provides a usage guideline for the stereotyped elements. The guideline states that a persistent class can only be associated with other persistent classes (if the owned end is navigable). The usage guideline is expressed as a constraint. The definition of the stereotype is presented in Fig.6.

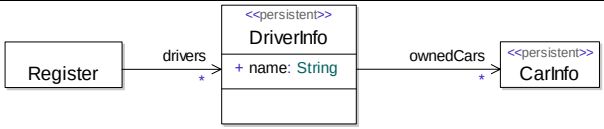
Name & representation	«persistent»	
Description	Stereotype used to designate that objects of this class should be stored in a persistent storage	
Tag definitions	isStoredInDB: Boolean	The tag isStoredInDB indicated whether objects of this class should be stored in a database or not. If isStoredInDB is set to true, the objects should be stored in a DB; if it is set to false, then the objects should be serialized to a file
Constraints	c1	Persistent classes can only be related to other persistent classes (if the association is navigable from the stereotyped class).
Example usage		

Fig. 6. Example of MS_3 stereotype - «persistent»

The presented kinds of model simplification stereotypes MS_1-MS_3 gradually provide more possibilities of customizing the base language. In general the model simplification stereotypes, however, are intended to make the designs simpler by using the stereotypes. They might be targeted to emphasize certain properties of designs or designate “specificity” of stereotyped elements. They are not intended to introduce changes to the modeling language in the same sense as a change to a metamodel would be introduced – with the intention to “enrich” the vocabulary of modelers. The model simplification stereotypes, however, should not be used as another way of showing inheritance – this would simply be a misuse of a stereotype.

3.2. Virtual metamodel extension stereotypes

Virtual metamodel extension stereotypes are stereotypes that are intended to provide new modeling constructs for modelers thus enriching the vocabulary available to them. Virtual metamodel extension stereotypes are usually defined as part of profiles intended to create a new "dialect" of UML by providing new modeling constructs. Therefore these stereotypes are advised to be created in a more rigorous way that model simplification stereotypes. There are, however, different kinds of virtual metamodel extension stereotypes to choose from. The choice depends on the purpose for which the stereotype is defined and is supported by the following decision graph:

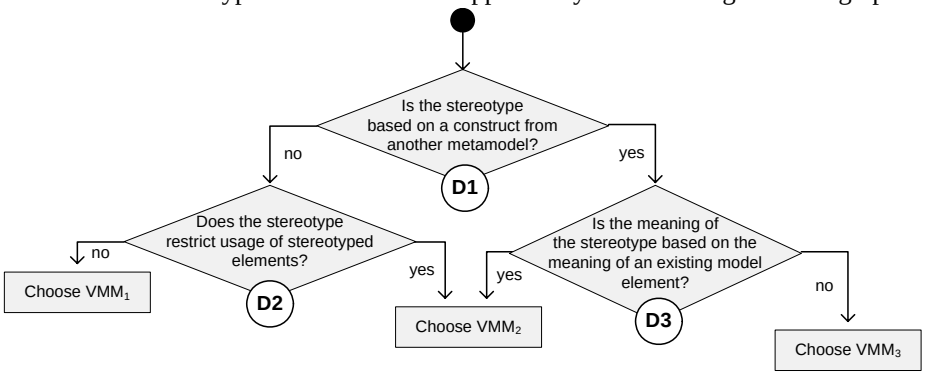


Fig. 7. Choosing between virtual metamodel extension stereotypes

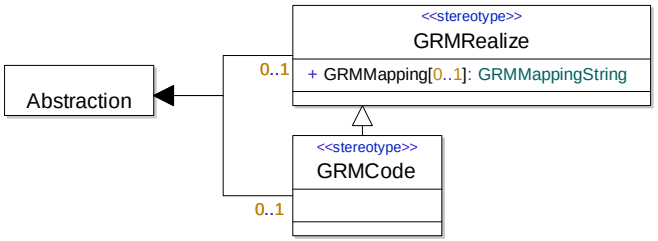
Virtual metamodel extension stereotypes can be defined if the stereotypes are used to mimic constructs from another (than UML) metamodel – further referred to as the defining metamodel since it defines the structure of a set of stereotypes (i.e. in the profile). The defining metamodel describes new constructs and relationships between them which should be added to the UML metamodel. If metamodeling is not possible, metaclasses in the defining metamodel can be "translated" into stereotypes and relationships between the constructs should be "translated" to allowed relationships between stereotypes (the UML specification allows associations between stereotypes if there exists associations between the model elements which the stereotypes extend – c.f. [2]). An example of a defining metamodel is the Software Process Engineering Metamodel (SPEM [22]) which is used as a basis for creation of a UML profile corresponding to the metamodel.

The lightest of the virtual metamodel extension stereotypes are stereotypes which are not defined based on the defining metamodel (D1) and they are not meant to restrict the usage of the stereotyped element (D2). Such stereotypes (VMM_1):

- Should be created by modelers. They should be created together with other stereotypes as auxiliary constructs – stereotypes of this kind are should not be used as the most important stereotypes in profiles which contains them.
- Should contain tag definitions that add properties to the extended model element.
- Should contain no constraints restricting the usage of the stereotyped model element.
- Can contain icons, but usually the icons are not used for this kind of stereotypes.

- Documented in documents which contain:
  - Name and representation of the stereotype
  - Description of the stereotype (i.e. intention and meaning of the stereotype)
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.
  - Relations to other stereotypes.
  - Examples of using the stereotype and the stereotyped elements.

An example of VMM_1 stereotype is the stereotype «GRMCode» which is defined as part of the UML Profile for Schedulability, Performance and Time [23].



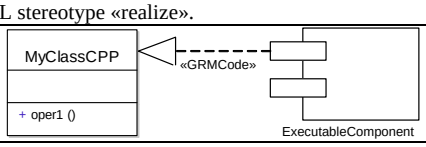
Name & representation	«GRMCode»	
Description	A mapping from a model element whose corresponding behavior is specified by a program of some type (e.g. Classifier) to either a UML component or to a node. This mapping is used to denote that the client physically contains the program code for the supplier.	
Tag definitions	GRMMapping[0..1]: GRMMappingString	The mapping tagged value should only be used if the mapping details are not fully specified by the realization relationship itself and the details are relevant to the analysis at hand
Related to	Inherits from the stereotype «GRMRealize» which inherits from the standard UML stereotype «realize».	
Example usage		

Fig. 8. Example of VMM_1 - «GRMCode»

Stereotypes that allow making the modeling language more precise are virtual metamodel stereotypes of the second kind (VMM_2). These stereotypes should be used either when:

1. They should represent constructs from the defining metamodel (D1) which are similar to some existing modeling elements (D3), or
2. They do not represent constructs from the defining metamodel (D1) but they restrict the usage of stereotyped elements (D2).

These stereotypes:

- Should be created
  - in a dedicated process with thorough verification and validation.
  - by experienced modelers
- Can contain icons, but usually no icons are necessary for these stereotypes.
- Can contain tag definitions adding properties to extended model elements.

- Should contain constrains restricting the usage of the stereotyped model elements.
- Be documented in a way which contains:
  - Name and representation of the stereotype
  - Excerpt from the defining metamodel
  - Description of the stereotype (i.e. intention and meaning of the stereotype)
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.
  - Description of the intention, meaning of the constraints defined for the stereotype. The constraints should be expressed both in OCL (or another language which is used in automatic checking of constraints on models) and in natural language (to increase understanding of the constraints by modelers).

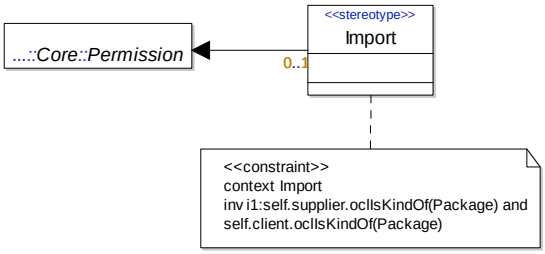
The meaning (semantics) of these stereotypes should make the semantics of the extended model element more precise and detailed, but it should not "redefine" the meaning of the extended model elements completely. An example VMM_2 stereotype is the stereotype «Import» from the SPEM profile.

If the stereotype is defined based on a construct in the defining metamodel (D1) and the meaning of the construct changes the meaning of the extended model element (D3), then the most powerful kind of virtual metamodel extension stereotypes should be chosen (VMM_3). These stereotypes:

- Should be created
  - In a dedicated development process with rigorous verification and validation methods
  - By experienced method specialists
  - As part of profiles aimed at creating a new "dialect" of UML
- Can contain icons which represent the concrete syntax of the construct in the defining metamodel.
- Can contain tag definitions which add properties of the construct from the defining metamodel.
- Should contain constraints which allow using stereotyped elements only with other stereotyped elements (both stereotypes should represent constructs in the defining metamodel for which the stereotype is defined).
- Be documents in a document which contains:
  - Name and representation of the stereotype
  - Description of the stereotype (intention and meaning)
  - Fragment of the defining metamodel which defines this construct
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.
  - Description of the intention, meaning of the constraints defined for the stereotype. The constraints should be expressed both in OCL (or another language which is used in automatic checking of constraints on models) and in natural language (to facilitate understanding of the constraints by modelers).

The meaning of the stereotyped elements instantiated in user models is different from the meaning of the base model elements. This allows changing (or tweaking) the UML to suit specific needs of the company.

An example of VMM_2 stereotype is «Import» from the SPEM profile.



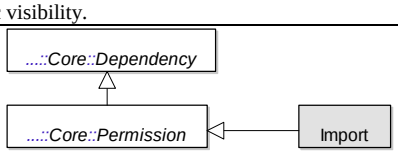
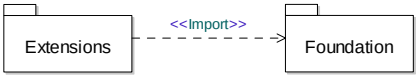
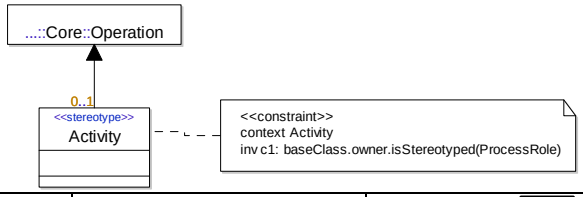
Name & representation	«Import»	
Description	An import dependency denotes that the contents of the target Package are added to the namespace of the source Package. This has similar semantics as UML Import except that in SPEM all elements have public visibility.	
Defining metamodel fragment		
Tag definitions	n/a	
Constraints	i1	The supplier and client must be kinds of Package: context Import inv i1:self.supplier.oclIsKindOf(Package) and self.client.oclIsKindOf(Package)
Related to	n/a	
Example usage		

Fig. 9. Example of VMM_2 - «Import»

An example of VMM_3 stereotype is «Activity» from the SPEM profile [22, p. 11-8]. Its definition is done in a similar format as the definition of «Import» except for the fact that the semantics of the stereotyped element is different from the semantics of the base model element (Operation). «Activity» stereotyped operations are intended to define activities performed in a software development process and they are allowed to be used only with certain stereotyped model elements – which makes the stereotype a VMM_3 stereotype. For the sake of simplicity of the example we omit several constraints, which are specified in a similar manner as i1 in Fig. 10.



Name & representation	«Activity»	
Description	*Activity* describes a piece of work performed by one ProcessRole: the tasks, operations, and actions that are performed by a role or with which the role may assist. An Activity may consist of atomic elements called *Steps*.	
Defining metamodel fragment	...::Core::Operation ← WorkDefinition ← Activity	
Tag definitions	n/a	
Constraints	i1	Every Activity is owned by a ProcessRole: context Activity inv c1: baseClass.owner.isStereotyped(ProcessRole)
Related to	«ProcessRole»; «Step»; «WorkDefinition»	
Example usage	Developer (ProcessRole) — Create Use Case diagrams and specifications	

Fig. 10. Example of VMM_3 - «Activity»

3.3. Code generation

A specific use of stereotypes is using stereotypes to improve code generation from models. Stereotypes dedicated for this purpose are usually defined in profiles aimed at providing a support for a specific programming language in a UML tool. There are, however, two kinds of code generation stereotypes which a distinguished by the degree of modification they introduce into the base model element. Choosing between the kinds of code generation stereotypes is supported by the decision graph:

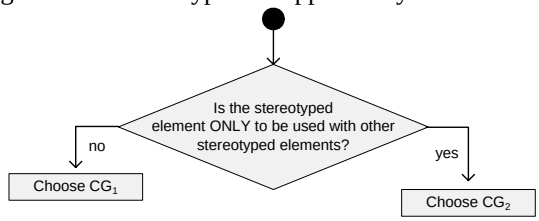


Fig. 11. Choosing between code generation stereotypes

If the stereotyped elements can be used only with other stereotyped elements then the degree of modification that they introduce should be substantial. The stereotypes (CG_2) should restrict the usage of the stereotyped model elements to such extent that it can actually be seen as a “redefinition” of the standard model element – the

stereotyped elements are specifically provided with context in which they should be used (i.e. explicitly stated which elements they can be used with). These stereotypes:

- Should be created
  - as part of a process of tool customization
  - by developers and modelers who work with providing support for new programming language in the modeling tools they use
- Should contain tag definitions which add properties which are required for code generation. The properties are added so that the code generators have all information which is required to generate more than just a skeleton source code.
- Should contain constraints (written in OCL or other language which can be automatically evaluated) which allow using stereotyped elements only with other stereotyped elements (both stereotypes should represent constructs in the programming language for which the code is to be generated). The constraints should forbid modelers from creating models which cannot be translated into source code. This is a crucial issue if the stereotypes and profiles are intended to realize the vision of model driven software development (c.f. [10]).
- Be documents in a document which contains:
  - Name and representation of the stereotype
  - Description of the stereotype (intention and meaning)
  - Excerpt of source code which this stereotype should represent.
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.
  - Description of the intention, meaning of the constraints defined for the stereotype. The constraints should be expressed both in OCL (or another language which is used in automatic checking of constraints on models) and in natural language (to facilitate understanding of the constraints by modelers).
  - Code generation rule or template which defines how the code is to be generated.

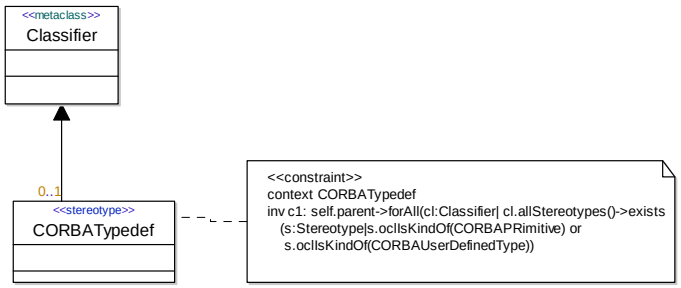
The meaning of the stereotyped element might be different from the meaning of the base model element. An example of CG_2 stereotype is «CORBATypedef» from the UML profile for CORBA [4].

Another kind of code generation stereotype is less strict and makes the stereotyped model elements only a more precise version of the standard model elements. These stereotypes (CG_1) should be chosen when the stereotyped elements should be used together with standard model elements (i.e. when the customization for the programming language construct is only making the model elements more precise so that the code generation results in a more complete code). These stereotypes:

- Should be created
  - as part of a process of tool customization
  - by developers and modelers who work with providing support for new programming language in the modeling tools they use
- Should contain tag definitions which add properties which are required for code generation. The properties are added so that the code generators have all information which is required to generate more than just a skeleton source code.
- Should contain constraints (written in OCL or other language which can be automatically evaluated) which forbid using stereotyped elements with certain

elements. The constraints should forbid modelers from creating models which cannot be translated into source code.

- Be documents in a document which contains:
  - Name and representation of the stereotype
  - Description of the stereotype (intention and meaning)
  - Excerpt of source code which this stereotype should represent.
  - Description of the intention and meaning of the tag definitions which are part of the stereotype.
  - Description of the intention, meaning of the constraints defined for the stereotype. The constraints should be expressed both in OCL (or another language which is used in automatic checking of constraints on models) and in natural language (to facilitate understanding of the constraints by modelers).
  - Code generation rule or template which defines how the code is to be generated.



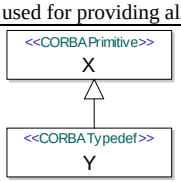
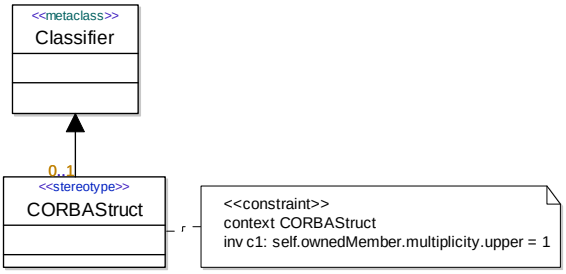
Name & representation	«CORBATypedef»	
Description	The stereotypes allow modeling an *IDL typedef* construct which is used for providing aliases for datatypes.	
Example excerpt of the source code		typedef X Y;
Tag definitions	n/a	
Constraints	c1	The parent must be stereotyped as «CORBAPrimitive» or as a descendant of «CORBAUserDefinedType» context CORBATypedef inv c1: self.parent->forAll (cl:Classifier| cl.allStereotypes()->exists (s:Stereotype| s.oclIsKindOf(CORBAPRimitive) or s.oclIsKindOf(CORBAUserDefinedType))
Code generation template	typedef {*NameOfStereotypedElement*} {*NameOfParentOfTheElement*};	

Fig. 12. Example of CG_2 - «CORBATypedef»

Stereotypes of this kind should make the existing model elements more precise in the context of models aimed for code generation. The stereotyped elements, nevertheless, do not change the meaning of the base model element. This leads to an observation based on investigating the set of profiles for improving code generation.



Name & representation	«CORBAStruct»	
Description	The stereotype represents *IDL struct* definitions.	

Example excerpt of the source code	<<CORBAStruct>> Structure1 field1: int	struct Structure1 {
		int field1;
		}

Tag definitions	n/a	
Constraints	c1	All navigable AssociationEnds must have the upper multiplicity value equal to 1 context CORBAStruct inv c1: self.ownedMember.multiplicity.upper = 1

Code generation template	struct {*NameOfStereotypedElement*} {
	for each attribute
	{*typeOfAttribute*} {*NameOfAttribute*};
	}	

Fig. 13. Example of CG_1 - «CORBAStruct»

The profiles which intend to improve code generation of general models contain stereotypes CG_1 which make the stereotyped elements more precise. The profiles which are intended to make UML suitable to model "code" (i.e. the aim of models created with this profile is to be a very detailed model which is aimed at [almost] 100% code generation) contain stereotypes CG_2. The latter profiles are substantially more difficult to create and should be created by experienced methods and tools specialists.

4. Conclusions

This paper presents a set of guidelines on how to create stereotypes which are fit for the purpose there are intended for. The previously elaborated purposes were organized in classifications of stereotypes which were used as a basis for our studies. The needs for these guidelines were identified in our previous industrial case studies in the context of realization of a vision of effective modeling – i.e. model driven software development.

The guidelines presented in this paper are intended to improve the practice of using stereotypes in industry by introducing eight different types of stereotypes. Various levels of documentation and different elements are associated with each type in order to differentiate between purposes and intentions of stereotypes. The simplest

stereotypes are advised to be created in a simple and informal way in order to decrease the time required for creating profiles containing these stereotypes. The most advanced purposes of stereotypes require including more advanced elements in definitions of stereotypes and hence entail more rigorous process of their creation and more detailed documentation. Using these guidelines allow increasing awareness of the issues related to language customization and provide means for effective creation of language dialects with specialized profiles.

In our future work we intend to evaluate these guidelines in an empirical form by verifying the actual gains from using the guidelines in an industrial setting. We are also working on a tool support for automatic generation of specification documents (e.g. tables) presented in this paper in order to assist stereotype creators in their work.

References

1. Miller J. and Mukerji J., "MDA Guide", Object Management Group, 2003, www.omg.org/mda/, last accessed 2005-01-10.
2. Object Management Group, "Unified Modeling Language Specification: Infrastructure Version 2.0", Object Management Group, 2004, www.omg.org, last accessed 2004-02-20.
3. Object management Group, "UML 2.0 Superstructure Specification", Object Management Group, 2004, www.omg.org, last accessed 2005-03-31.
4. Object Management Group, "UML Profile for CORBA", Object Management Group, 2002, www.omg.org, last accessed 2003-09-20.
5. Wirfs-Brock R., "Stereotyping: A Technique for Characterizing Objects and Their Interactions", *Object Magazine*, vol. 3, 1993, pp. 50-3.
6. Staron M., Kuzniarz L., and Wallin L., "A Case Study on Industrial MDA Realization - Determinants of Effectiveness", *Nordic Journal of Computing*, vol. 11, 2004, pp. 254-278.
7. Kuzniarz L., Staron M., and Wohlin C., "An Empirical Study on Using Stereotypes to Improve Understanding of UML Models", In the Proc. of The 12th Int. Workshop on Program Comprehension, Bari, Italy, 2004, pp. 14-23.
8. Staron M., Kuzniarz L., and Wohlin C., "An Industrial Replication of an Empirical Study on Using Stereotypes to Improve Understanding of UML Models", In the Proc. of Soft. Eng. Research and Practice in Sweden, Linköping, Sweden, 2004, pp. 53-62.
9. Yilmaz C., Memon A. M., Porter A. A., Krishna A. S., Schmidt D. C., Gokhale A., and Natarajan B., "Preserving Distributed Systems Critical Properties: A Model-Driven Approach", *IEEE Software*, vol. 21, 2004, pp. 32-40.
10. Staron M., Kuzniarz L., and Wallin L., "A Case Study on Transformation Focused Industrial MDA Realization", In the Proc. of 3rd UML Workshop in Soft. Model Eng., Lisbon, Portugal, 2004.
11. Staron M. and Kuzniarz L., "Properties of Stereotypes from the Perspective of Their Roles in Designs", In the Proc. of 8th Int. Conf. on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, 2005, accepted for publication.
12. Gogolla M. and Henderson-Sellers B., "Analysis of UML Stereotypes within the UML Metamodel", In the Proc. of «UML» 2002, Dresden, Germany, 2002, pp. 84-99.
13. Berner S., Glinz M., and Joos S., "A Classification of Stereotypes for Object-Oriented Modeling Languages", In the Proc. «UML» 1999, Fort Collins, USA, 1999, pp. 249-64.
14. Object Management Group, "Unified Modeling Language Specification V. 1.4", Object Management Group, 2001, last accessed 2003-05-30.
15. Kuzniarz L. and Staron M., "On Practical Usage of Stereotypes in UML-Based Software Development", In the Proc. of Forum on Design and Spec. Languages, Marseille, 2002, pp. 262-270.

16. Atkinson C., Kühne T., and Henderson-Sellers B., "Stereotypical Encounters of the Third Kind", In the Proc. of «UML» 2002, Dresden, Germany, 2002, pp. 100-14.
17. Atkinson C., Kühne T., and Henderson-Sellers B., "Systematic Stereotype Usage", *Software and Systems Modeling*, vol. 2, 2003, pp. 153-163.
18. Object Management Group, "Meta Object Facility (MOF) Specification V. 1.4", Object Management Group, 2001, www.omg.org, last accessed 2003-10-08.
19. Atkinson C. and Kühne T., "Rearchitecting the UML Infrastructure", *ACM Transactions on Modeling and Computer Simulation*, vol. 12, 2002, pp. 290-321.
20. Wirfs-Brock R., Wilkerson B., and Wiener L., "Responsibility-Driven Design: Adding to Your Conceptual Toolkit", *ROAD*, vol. 2, 1994, pp. 27-34.
21. Henderson-Sellers B., "The Use of Subtypes and Stereotypes in the UML Model", *Journal of Database Management*, vol. 13, 2002, pp. 43-50.
22. Object Management Group, "Software Process Engineering Metamodel Specification", OMG, 2005, www.omg.org, last accessed 2005-05-01.
23. Object Management Group, "UML Profile for Schedulability, Performance and Time", Object Management Group, 2002, www.omg.org, last accessed 2003-09-20.

Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams

Sven Wenzel

Tampere University of Technology, Finland
University of Dortmund, Germany
email@svenwenzel.com
http://www.svenwenzel.com

Abstract. An approach for the detection of structural patterns based on UML class diagrams is presented. By using a fuzzy-like evaluation mechanism the introduced approach is able to recognize not only entire patterns but also incomplete instances.
Referring to structural patterns in general the knowledge about used patterns assists a developer not only while maintaining or reverse engineering existing software, but already while designing or implementing new software. The information about the instantiation status is essential for a developer using, for example, specialization patterns that guide the extension of a particular framework. The developer can be supported by information about already instantiated patterns as well as partial instances which obviously occur rather often while developing.

1 Introduction

Since software systems become larger and more complex, the task of understanding while developing and especially while maintaining software becomes more and more difficult. Therefore the use of patterns has become a helpful methodology to develop software in a more structured and understandable way.

In general, a pattern is a scheme that consists of three main parts: a context, a problem, and a solution. They discuss a particular recurring problem which arises in a particular situation – the context. Furthermore they offer a proven solution to this problem.

There exist several pattern families which focus on different aspects of software development and take place in the different stages of the development process. The main interest within the design and implementation phases is directed towards those patterns that focus on problems of software design. These are behavioral patterns that focus on the run-time behavior of software elements and structural patterns that concentrate on the structural arrangements of software elements. Both types can be separated into more abstract and more concrete problems.

Design patterns [1] are more abstract and focus on problems in object-oriented software in general. For example, the *Composite Pattern* describes how to compose several objects into a part-whole hierarchy with a uniform interface.

Specialization patterns [2] do not discuss general project-independent problems, but focus on more specific topics of concrete projects. They describe, for example, how to extend a particular framework and support users in this process.

From the field of design pattern detection it is well-known that the knowledge about used patterns in a software helps the developer to get a better understanding of it (see Section 5, Related Work). Especially in maintenance and reverse engineering the occurrence of the solution parts of particular patterns help the developer to understand the original problems.

But the knowledge about used patterns – especially with regard to structural patterns in general – assists the developer already while designing or implementing new software. As in maintenance and reverse engineering, the reason is a better understanding, but the circumstances are quite different. Since the software is currently in development, the likelihood of partly instantiated patterns is rather high. However, the knowledge about those incomplete patterns helps the developer and should not be neglected.

An example is the use of specialization patterns guiding a developer while extending a framework. The patterns define the classes that have to be created, the interfaces that have to be implemented, or the operations that have to be overwritten, etc.. Furthermore, some architectural rules may be defined as a pattern to enforce structural properties of the developed software. The developer would like to check if her design satisfies these architectural rules and she also wants to have information about the work progress, in other words, the tasks that have to be done.

Consequently, an approach for detection of structural patterns should be able to recognize these incomplete instances as well as entire ones. The approach introduced on the next pages provides an insight into a possible realization of the detection of incomplete patterns based on the Unified Modeling Language (UML) [3] that has become a widely accepted standard for designing software in academia and industry. It uses a combined bottom-up/top-down analysis and a fuzzy-like mechanism to evaluate a given UML model.

The following section introduces the pattern concept used in this approach to be more independent from specific pattern types as design patterns or specialization patterns. Section 3 presents the recognition approach itself and introduces some characteristics of incomplete pattern instances. The implementation of this approach is discussed in Section 4. Finally, Section 5 summarizes related approaches for pattern detection and Section 6 summarizes current and future work.

2 Pattern Definition

This approach describes and expresses patterns in a more general way to be more independent from any specific pattern types, such as design patterns or specialization patterns.

Detached from the pattern type, the solution part of a structural pattern defines an arrangement of software elements to solve a particular problem. Since

the problem itself is not of interest here, the arrangement of software elements is formalized for the search in a neutral manner regarding to the pattern type.

As this arrangement is in fact rather a template than a combination of concrete software elements, these template elements are called *roles*. Roles are placeholders that can be taken from concrete elements in the instance of the pattern. Each role has a type (e.g., classifier or association) to determine the kind of software elements that can act as the role. Since software elements allow the nesting of other elements, each role may contain several subroles representing nested elements.

However, the existence of roles and their nesting relations in between is not sufficient to express complex arrangements of software elements, so that roles can be enhanced by constraints. These constraints are given in the Object Constraint Language (OCL) [3, Chap. 6] and enforce certain properties of the concrete elements acting as the role. They define, for example, visibility or stereotype properties. Furthermore they may refer to other roles to express particular relations like inheritance or parameter types.

By default every role has to be played exactly once in a pattern instance, but it is possible to define multiplicities to give limitations for the amount of elements acting as a role in a pattern instance. The multiplicity provides a lower and an upper range as it is done for association ends in UML. A lower range of zero makes a role optional and an infinite upper range allows as many elements acting the role as possible.

An example for the neutral representation is shown for a design pattern in Figure 1. Each element of the UML class diagram is translated to a role. The type of the UML element determines the type of the role. Child elements (e.g., parameters of operations) become subroles and properties of elements (e.g., abstraction or inheritance) are replaced by constraints for the corresponding role.

The graphical notation for the pattern definitions used in this article is a UML object diagram extended by some features of UML class diagrams. Object nodes represent roles – labeled with the name and the type of the role, separated by a colon. Aggregations express containments of subroles and constraints are represented by notation elements. For dependencies between roles dashed arrows are used.

3 Automatic Detection

The previously presented notation for a structural pattern based on a UML object diagram already describes exactly what to search for. The developer wants to find an object of the type class that acts as the context role (cf. Fig. 1). Furthermore this class object should contain an operation object that acts as the register operation. This operation object again should contain a parameter and so on.

This situation of picking elements for particular roles can be compared to a casting for a theater play. Result of the search is a set of mappings between particular roles and the elements acting as these roles, whereas the elements are

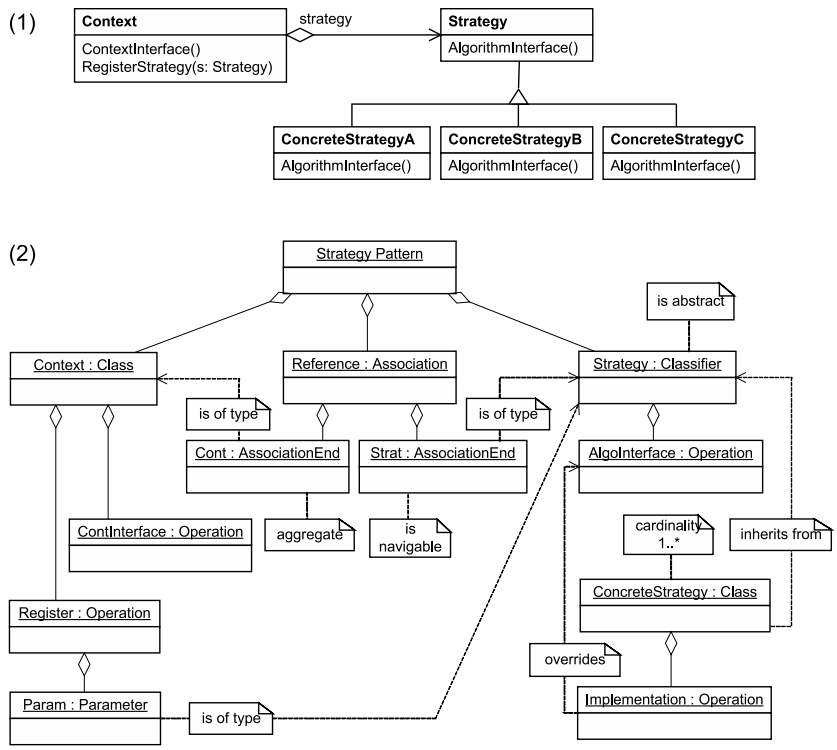


Fig. 1. The design pattern Strategy pictured as **(1)** a UML class diagram (from [1]) and **(2)** as an independent pattern definition. In this example the pattern is extended by an operation for registering a strategy.

called *candidates* and the whole set is called a *cast*. The single mapping between one role and a candidate for this role is called a *binding*; it binds a candidate to its role.

Each binding is associated with a quality value that expresses how well the candidate acts as the role. As in the world of theater there will exist several candidates for a particular role or one candidate for several roles, however, in each case the role is treated with a different quality.

If the developer was interested only in complete pattern instances, she would select only those candidates that act their role par excellence. In the case of incomplete instances the developer cannot find perfect candidates for each role and those candidates with minor quality become interesting. Therefore the algorithm does not bind candidates to roles by simple *yes-or-no* decisions but rather uses assignments with intermediate values as it is usual in fuzzy-logic.

These intermediate values are expressed in the quality of a binding. The quality values range between 0% and 100%. The value is zero, if the candidate

cannot act as the role at all; 100% means that the candidate satisfies all of its requirements. The value itself is calculated by the constraints and the subroles.

Generally it is possible to create every binding right from the start, because if there is a role of type class, every class could be a candidate for this role. Even if all constraints and subroles are unsatisfied, it is still a class and consequently a candidate for this role. This technique will obviously end up in an unmanageable set of bindings and has to be more organized.

The detection algorithm is operating in three major phases. In a first phase the roles are classified into a hierarchy to improve the binding process. Then in the consecutive phase this hierarchy is traversed in bottom-up direction to locate all candidates for each role. Finally, in a top-down phase the best candidates of the searched pattern are selected.

3.1 Classifying Phase

The definition of the pattern is a graph of roles connected by child or dependency relationships. The child relationships can also be considered as dependencies because an element should act as a particular role only if also its children act as the corresponding subroles.

Hence, the graph can be sorted by those dependencies to organize the roles into different levels (see Fig. 2), so that each role is assigned to a particular level. The roles that have no dependencies to other roles are assigned to the lowest level. Those roles with dependencies to other roles are assigned to the next level above the highest level of all supplying roles. Thereby, the lowest level contains all roles that have no dependencies to other roles. Every next level contains all roles that have dependencies only to roles from the lower levels. The highest level contains only one virtual role that represents the pattern itself. It depends on its children – the main roles of a pattern.

The generated hierarchy is the basis for the detection. If the hierarchy is processed level-by-level from the bottom to the top as it is done in the bottom-up phase, it is ensured that supplying roles are always checked before their clients.

3.2 Bottom-Up Phase

The bottom-up phase searches the candidates for each role. It starts with the roles at the lowest level of the beforehand generated hierarchy, as they are independent from any other role.

First, the algorithm locates all elements from the model, which can act as the particular role by their type. Thus, every operation of the entire model could act as the role *AlgoInterface* (cf. Fig. 2) and for each of those operations a temporary binding is created. Indeed, this procedure will cause a huge amount of bindings and might be inefficient for large models. However, this lack of efficiency can be disregarded here, because it is assumed that software models have a natural limitation of elements as they are developed by human beings.

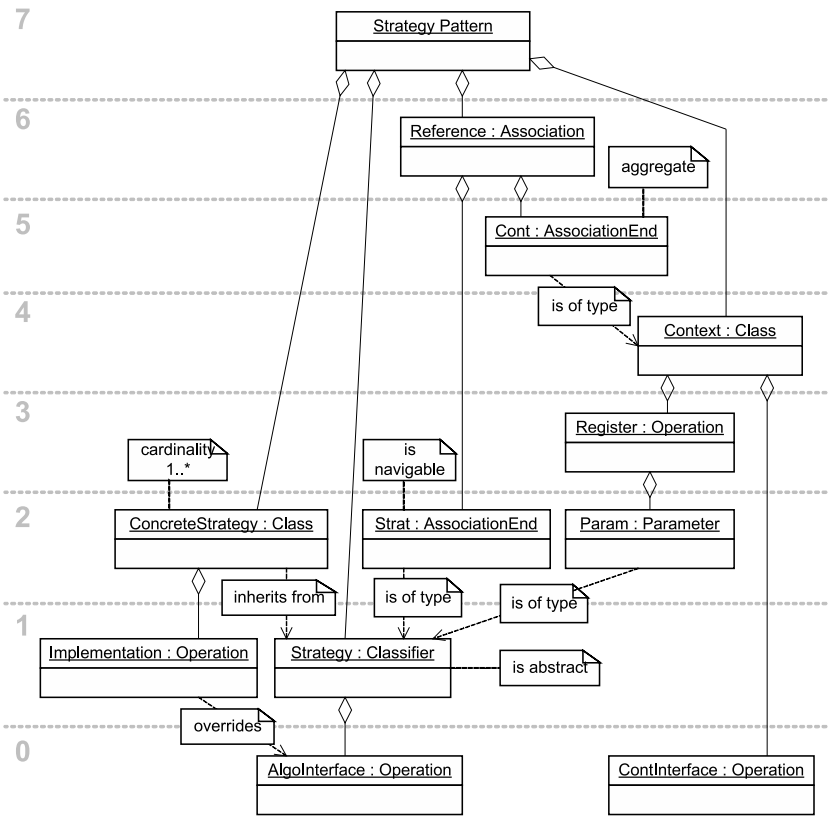


Fig. 2. The ordered role graph of the Strategy pattern. The roles are arranged on different levels based on their dependencies. Those dependencies are either caused by constraints or parent-children relationships.

Once a role is temporarily bound to elements, the constraints assigned to the role are evaluated for the bound element, because the element type is not the one and only criterion for bindings. The evaluation of a constraint results in a value between 0% and 100% expressing how well the constraint is satisfied. The value of a simple constraint, e.g., *"is abstract"*, is 100% in case of satisfaction, 0% otherwise. However, it is possible that a more complex constraint returns intermediate values if it is satisfied partially.

If a role of the lowest level has no constraints at all, there are no constraints to assess the binding and it is taken as fully satisfied. Otherwise the quality of the binding is reflected by the average quality of the constraints. Thus the mean of all constraint evaluation results is calculated and taken as the quality of the

binding.

$$q(r) = \frac{1}{|C|} \cdot \sum_{i=1}^{|C|} eval(C_i) \quad \text{if } |C| > 0 \tag{1}$$

The accuracy of the evaluation can therefore be increased by defining as much constraints for each role as possible. The more constraints specify the characteristics of a role the more precise the quality of a binding is determined. The abandonment of constraints at all does consequently annul the quality calculation.

Constraints of roles from all levels higher than zero may depend on other roles. In this case the returned value of the constraint evaluation is dependent on the quality of the binding referred to by the constraint. An example is the constraint evaluation of *Implementation* (cf. Fig. 2): The *override* constraint defines that the operation bound to *Implementation* has to override an operation bound to *AlgoInterface*. The evaluation returns 0% if the operation found does not override another operation at all or the overridden operation is not bound to *AlgoInterface*. Otherwise it returns the quality of the binding between the overridden operation and the role *AlgoInterface* which should be overridden here. Since this binding has a particular quality, the constraint is satisfied by this quality.

Furthermore, the roles from all levels higher than zero may contain also subroles. Each of those subroles, S_i, can be interpreted as a constraint *"has a child of type S_i"*. Consequently, the calculation of the quality is extended by those subroles S, which are handled in the same manner as additional constraints.

$$q(r) = \frac{1}{|C| + |S|} \cdot \sum_{i=1}^{|C|} eval(C_i) + \frac{1}{|C| + |S|} \cdot \sum_{i=1}^{|S|} q(S_i) \tag{2}$$

These subrole constraints are calculated like those constraints depending on other roles – here, the subroles. If no candidate for the subrole (i.e., a child element in the model) is found, the constraint is treated as 0%. Otherwise it has the quality of the binding of the candidate found.

The algorithm traverses upwards the role graph from level to level as described above. For each role new temporary bindings are created and evaluated. All unsatisfied bindings (i.e., those with a quality of 0%) are deleted immediately. The other bindings, which satisfy at least some constraints or subrole constraints, are stored in a cast for the later top-down phase.

To make this calculation phase more efficient, the threshold for keeping and deleting bindings can optionally be changed from zero to any other value. For example, to 50% to keep only bindings which are half satisfied. Choosing 100% would yield a detection of complete patterns.

The result of the bottom-up phase is a set of bindings from roles to different elements. Each binding has a particular quality that expresses how well the role is acted by its element.

3.3 Top-Down Phase

The detection algorithm switches to the top-down phase, when the virtual role representing the pattern is reached (i.e., the role of the highest level). The cast generated so far contains a lot of bindings that are not necessarily part of a pattern instance, especially if the threshold was set to a small value or zero.

These bindings are called *false bindings* and are filtered out in this phase of the algorithm. Therefore the pattern definition graph is now handled as a tree by taking the child relationships as edges and the virtual role of the pattern as the root. The dependencies are not taken as edges, but for additional information only.

The graph is now traversed downwards in a breadth-first search order. For each role the binding with the highest quality is selected to be kept in the final cast. In case of equal qualities of multiple bindings for a role it is checked which binding is supplier for other bindings. Bindings with more clients than others are preferred. Also the information about other roles acted by the element of the binding is analyzed. It might be that an element that acts as the current evaluated role is the only candidate for another role. Especially if an element is supposed to act as only one role at a time, it has serious consequences for which role the element is taken.

During the entire procedure the child relationships of the actual elements in the investigated model are considered. Only the children of a particular element in the model can act as the subroles of the role of that element. Furthermore the role multiplicities are considered in a way that roles with an upper range greater than one allow the selection of a suitable amount of bindings.

The top-down phase results in the cast that contains all bindings representing an instance of the searched pattern.

3.4 Several Pattern Instances

To simplify matters the previous introduction of the algorithm has skipped the support for finding multiple pattern instances. If taken as presented, the algorithm would just result in one found pattern instance. It would be the most satisfied one; however, the developer is interested in all instances.

The latter phases of the search, bottom-up and top-down, are accompanied by an assignment procedure that maps the found bindings to possible pattern instances. This procedure works with respect to the fact that one binding can occur in several overlapping patterns. For example, a model contains different *Contexts* that use the same *Strategy*.

The bottom-up phase assigns each binding to one or more pattern instances. Each newly found binding either belongs to a new pattern instance that has not been touched so far, or it belongs to one that has already other bindings assigned. Thus, each binding is assigned to a new pattern instance, or, if the binding has relationships to other bindings (i.e., children or dependencies), it is assigned to the same instances to which those related bindings are assigned.

These assignments between bindings and pattern instances are later respected in the top-down phase. The phase is not performed only once, but for each possible pattern instance. The top-down analysis thereby consults only those bindings that are assigned to the currently investigated pattern instance independent from the possibility that the elements may participate in other pattern instances as well.

Once again, a predefined threshold ensures that dispensable pattern instances are deleted. For example, a binding has been assigned to a pattern instance but it has never been referred to by another binding; obviously this instance containing one binding only is not very promising. Of course, a threshold of 100% would just keep complete instances.

With respect to the assignments between bindings and pattern instances, the algorithm results in a set of casts for different pattern instances. Each found pattern instance is annotated with a quality value in the same way as the qualities of single bindings.

While detecting incomplete pattern instances it is obviously difficult to talk about false results, because even a single class can be an incomplete pattern instance. However, the amount of these very incomplete instances depends on the choice for the threshold in the last step of the algorithm, but the developer has to check the result manually to decide which pattern instances produce interest.

4 Implementation

The previously introduced approach has been implemented as a layered architecture to keep it as independent from any particular semantics as possible.

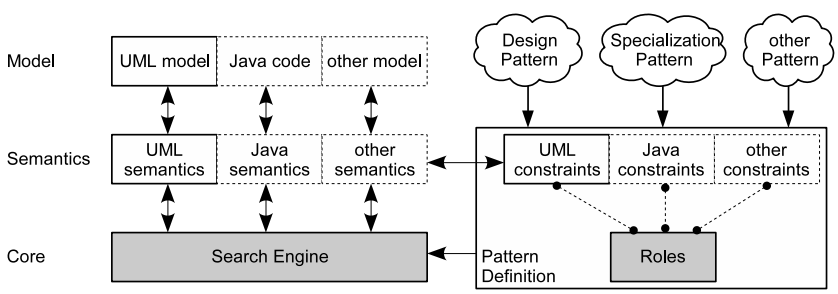


Fig. 3. The introduced approach has been implemented as a layered architecture.

As shown in Figure 3, the search engine itself takes just the pattern definition as an input. This definition consists of roles and constraints and is consequently independent from any specific pattern type. Only the constraints are related to the semantics of the investigated model, as they check, for example, aggregation

types that are part of the UML. The roles themselves are independent from semantics. They just know their relations to other roles and store the information on assigned constraints. Furthermore, they know the element type they can be bound to without knowing the semantics behind.

The search engine itself does not know any semantics about the models it analyzes. The bottom-up and top-down phases of the algorithm work independently from the investigated model. Each access to the model is encapsulated by an adapter that knows the semantics of the corresponding model. The evaluations of constraints are also processed by the semantic elements.

Thus, the core algorithm works just with semantic-less objects. The only assumption made by the algorithm is that the models on which it works have a compositional structure, so that the parent-children relations of the roles are reflected in the model.

So far, the semantics have been implemented of UML class diagrams. The *Eclipse UML2* project [4] provides the basis for representing the models and the implementation of the semantics layer works directly on these models. Most of the constraints have been realized on basis of OCL using the *Kent OCL Library* [5]. Simple boolean OCL expressions allow to check if a particular property is satisfied, e.g., `context NamedElement inv: visibility='public'`. Other OCL expressions can return an element of the model that is checked to be bound to a particular role, like `context Class inv: self.generalization.general=[Strategy]` to check inheritance. Naming and stereotype constraints use proprietary implementations that allow regular expressions.

5 Related Work

Since design patterns are the most used pattern type in implementation, there is an interest to detect them while maintaining or reverse engineering software. Thus, there exist many solutions noted for automated detection of design patterns.

Compared to this approach, the reverse engineering component of the FUJABA *Tool Suite* [6] is the most related one, because it gave some basic ideas. It uses graph grammars to operate on the abstract syntax graph (ASG) of a software system. Graph transformations are defined for each pattern and annotate the ASG with new nodes containing information about found instances. Therefore it uses alternating bottom-up and top-down phases and patterns are defined hierarchically as it is done with the roles in this approach. So far the FUJABA approach concentrates on structural patterns, but the support for behavioral patterns is already in development.

Heuzeroth et al. [7] present an approach based on predicate logic. It operates in two phases and is able to detect behavioral patterns as well. First, candidates for pattern instances are searched with the help of predicates in a static analysis. Second, the runtime behavior is analyzed to check if the expected behavior for the candidates is satisfied.

An approach based on software metrics is presented by Antoniol et al. [8]. It calculates class level metrics and compares them to a previously defined catalog of pattern metrics.

Keller et al. [9] deal with an intermediate representation of source code. It is held in a design repository that is based on a relational database and provides storage as well as querying of abstract design components that are in fact nothing else than pattern definitions.

Krämer and Prechelt [10] present an approach based on PROLOG. Both, design patterns and software designs, are expressed in PROLOG terms. The search for patterns is then done by the PROLOG engine.

An approach to detect design patterns with relational algebra is researched by Fronk and Berghammer [11]. Design patterns and structural information of a software system are expressed in relational terms. A relational calculator is used to solve the relational equations and recognizes the pattern instances.

6 Current and Future Work

The current work focuses on the integration of the search engine into *MADE* [12], an architecting tool that assists designers to define pattern-based architectures and developers to instantiate them. The knowledge of incomplete instances will assist the developer and allows the tool to process some tasks automatically.

As part of this integration it is aimed to allow a manual definition of bindings. Thus the users are able to assign elements to roles by themselves and the tool finds the related roles or reports their absence respectively. Furthermore, the integration of the new approach into a tool will allow empirical case-studies to prove the usability of this approach and to explore reasonable default values for the different thresholds.

One future objective will be the extension to semantics of programming languages, such as Java. This would increase the possible field of application of this approach, especially in the reverse-engineering.

However, the main future objective is continual improvement of the search engine itself. The possibility to assign single constraints with weights will allow a more precise pattern definition. More important will be support for behavioral patterns. In case of handling them as well as structural patterns the possible field of application will be increased much more.

Acknowledgements

The approach has been developed as part of the author's diploma thesis, written in cooperation between the Tampere University of Technology, Finland, and the University of Dortmund, Germany. The thesis is funded financially by the Martin-Schmeißer-Stiftung of the University of Dortmund.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Hakala, M., Hautamäki, J., Koskimies, K., Paaki, J., Viljamaa, A., Viljamaa, J.: Annotating reusable software architectures with specialization patterns. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam (2001) 171–180
3. The Object Management Group: Unified Modeling Language Specification – version 1.5 (formal/03-03-01). Online at http://www.omg.org/uml/ (2003)
4. Eclipse Foundation: Eclipse UML2 Project. http://www.eclipse.org/uml2/ (2005)
5. Akehurst, D., Patrascoiu, O.: Kent OCL Library. http://www.cs.kent.ac.uk/projects/ocl/ (2004)
6. FUJABA Tool Suite Developer Team - University of Paderborn: FUJABA Tool Suite. http://www.fujaba.de/ (2005)
7. Heuzeroth, D., Holl, T., Högström, G., Löwe, W.: Automatic design pattern detection. In: Proc. of the 11th IWPC'03, Portland, Oregon, USA (2003) 94–103
8. Antoniol, G., Fiutem, R., Cristoforetti, L.: Design pattern recovery in object-oriented software. In: Proc. of the 6th IWPC'98, Ischia, Italy (1998) 153–160
9. Keller, R., Schauer, R., Robitaille, S., Pagé, P.: Pattern-based reverse-engineering of design components. In: Proc. of the 21st ICSE'99, Los Angeles, California, USA (1999) 226–235
10. Krämer, C., Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: Proc. of the 3rd Working Conference on Reverse Engineering (WCRE'96), Monterey, CA, USA (1996) 208–215
11. Fronk, A., Berghammer, R.: Considering design problems in OO-software engineering with relations and relation-based tools. Journal on Relational Methods in Computer Science (JoRMiCS) **1** (2004) 73–92
12. Hammouda, I., Hautamäki, J., Pussinen, M., Koskimies, K.: Managing variability using heterogeneous feature variation patterns. In: Proc. of FASE'05, Edinburgh, UK (2005) 145–159

Task-driven Instantiation of Class Diagrams

Samuel Lahtinen, Imed Hammouda, Jari Peltonen, and Kai Koskimies

Tampere University of Technology, Institute of Software Systems,
P.O.Box 553, FIN-33101 Tampere, Finland
{samuel.lahtinen, imed.hammouda, jari.peltonen, kai.koskimies}@ tut.fi

Abstract. A well-defined modeling language is often defined by another model, the metamodel – typically presented as a class diagram. Basically a metamodel is the grammar definition of the language, and as such, it gives the rules and guidelines for how the models can be created. Especially in the case of complex metamodels, there is a need for tool support for creating valid models. In this paper we propose a method for task-driven instantiation of class diagrams. Metamodel is projected into a pattern which guides the user in the instantiation process. Using existing pattern tools, an environment can be generated for creating object configurations following the metamodel. We demonstrate the approach with a product configuration application. We conclude that the instantiation process can be made easier and less error prone by using a task-oriented instantiation process.

1. Introduction

The theory built around context-free languages was one of the success stories of software science in the 60's and 70's. This theory, and the mature technology relying on it, is to large extent behind the rapid development of modern programming languages and their tool environments, which are one of the cornerstones of modern software engineering. However, current software development technologies emphasize increasingly the role of various software models as abstract descriptions of systems. In contrast to traditional textual representations of software systems, the languages used for expressing such models usually cannot be defined using context-free grammars: models are typically represented as diagrams with complicated graph-like structure, rather than linear strings.

The standard way of specifying a modeling language is to use a metamodel, that is, another model that defines the rules for constructing models. An attractive approach to define a modeling language is to use the language itself, or its subset. This is the case for UML as well, the lingua franca of modeling languages for software systems. The main structural diagram type of UML, class diagrams (or their subset), is used to specify the structure of UML models, including class diagrams themselves. This specification is the metamodel of UML; the models are instances of the metamodel.

Unfortunately, our current understanding of metamodel-based language processing is far less mature than that of grammar-based languages. There is a lack of systematic and practically motivated approaches for creating, analyzing, and processing instances of metamodels, in a way comparable to the various technologies related to context-

free grammars. We anticipate that research on such approaches will become as necessary in near future – and hopefully as successful – as the research on context-free languages was in the 60's and 70's.

In this paper, we study a particular problem related to metamodel based modeling languages: how to support the creation of legal models, given a metamodel. This problem is analogous to the syntax-directed editing of textual languages, but it is even more important in the case of graphical modeling languages. In fact, most UML tools apply "syntax-directed editing" in the sense that they allow the user to create only models which comply with the UML metamodel, or at least aim to do so. In this way the parsing problem can be avoided, which is yet another insufficiently explored area in metamodel-based language processing. Thus, practical UML editors have solved the problem of supporting legal model construction, but each in its own, ad hoc way.

We study the problem of metamodel instantiation in a slightly more general context. UML offers another diagram type, object diagrams, to describe object configurations, for example, those created as instances of a class diagram. Thus, we can formulate the problem as follows: given a class diagram, how to support the creation of object diagrams that conform to the class diagram. If the class diagram happens to be the UML metamodel, the object diagram is an abstract representation of a class diagram. However, class diagrams can be used to specify the legal configurations of entities for any purpose, like the structure of products, documents, organizations, etc. In software engineering, class diagrams can be used, for example, in conceptual modeling, domain modeling, and detailed design of a system. Here we will use the specification of product configuration as an example application.

Our approach is to exploit existing tool support [Hau05, Ham05] to produce a task-driven environment for class diagram instantiation. This tool was originally developed for specifying and applying patterns in software systems. In our case, we view class diagrams as patterns, which are instantiated as object diagrams. This approach allows the user to create an object diagram conforming to a class diagram stepwise, following tasks generated by the tool. After all the compulsory tasks are completed, the user can be certain that a valid object diagram has been produced. The main advantage of the approach is that the task-driven process relieves the user from knowing the underlying metamodel. Especially in the case of complex class diagrams, the seemingly simple process of instantiating a class diagram and verifying the diagram created can be quite troublesome and error prone without proper tool support. Especially users who are unfamiliar with the details of the UML metamodel can easily make mistakes or miss some parts of the process.

The organization of this paper is as follows. Section 2 introduces the issues related to instantiation of class diagrams. In Section 3, a brief introduction to aspectual patterns is given. Section 4 explains the technique to create a pattern for instantiating a class diagram. In Section 5 the pattern tool applied in this work, MADE, is introduced and an example application of our approach is presented in Section 6. Section 7 discusses some related work and concluding remarks are presented in Section 8.

2. Instantiation of Class Diagrams

In a class diagram, we aim to name and classify the objects of a domain, as well as describe the features, structural relationships, and possibly other constraints among them. We aim to do this as accurate as possible, in order to define precisely the underlying domain. An instance of a class diagram presents an object configuration that captures the state of a system in a given time. These object configurations can change during time, but still, the instances must follow the rules given in the class diagram.

Since class diagram notation can be used for various purposes, the actual instance can be basically anything, including physical product configurations, documents, databases, software systems, etc. In UML, an instance of a class diagram can be presented as an object diagram. That is, a class diagram can be seen as an abstraction of an object diagram, having an infinite number of different object diagrams as its instances. Hence, the instantiation of a class diagram cannot be made fully automatic. It is the designer who has the knowledge to choose the desired object configuration for each instance. That is, the instantiation process can be seen as a guided "unfolding" (by giving values for the variables) of the class diagram.

According to OMG [OMG, section 3.20.], "An object diagram is a graph of instances, including objects and data values." Basically, the objects are instances of concrete classes, and links between the objects are the instances of the relationships between the classes. In addition, the attributes of classes have values in object diagrams. The multiplicities and other constraints, presented in class diagrams, do not have corresponding elements in the object diagrams, since they just limit the possible objects, values, and links in the object diagrams.

Abstract classes in class diagrams are merely a structuring mechanism, depicting that some classes can be classified further. Due to their abstract nature, abstract classes do not have directly corresponding instances in the domain, but any of the concrete classes inherited (either generalization or specialization relationship) from the abstract class can be instantiated instead. Otherwise, there are no limitations for the instantiation of a class in the description of the class itself. That is, there can basically exist any amount of objects as instances of a class, if there are no other constraints for the class.

In addition to specialization and generalization relationships, there are associations among classes. All the associations, including unidirectional, composite, and aggregate associations, are treated in the same way in the object diagrams, that is, they are shown as links between the objects. An association among classes determines the possibility to have links between the objects. To interpret the multiplicities, the associations are examined from the point of view of a single object (or its class). Then, the multiplicities in the other ends of the association determine the amount of other objects (of a type the classes determine) that can be linked to the object. That is, the associations can be used to determine the need for other objects.

The lower bound in the multiplicity range determines whether the link between objects is mandatory or not. If the lower bound is zero, the link between the corresponding objects is optional, otherwise it is mandatory. The upper bound determines the maximum amount of certain kind of objects in the relationship, but does not limit the amount of the objects of that type in general – just in that particular relationship. In their simplest form, multiplicities are fixed values instead of multiplicity ranges. Fig-

ure 1 A depicts such an association; for each object of class A there are exactly two objects of class B. The corresponding object diagram can be seen in Figure 1 B.

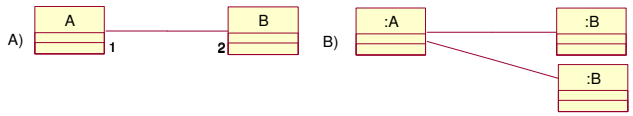


Fig. 1. Fixed multiplicity values of associations

As another example, let us consider the class diagram presented in Figure 2 A. For each object of class A there can be zero or one objects of class B and any number of D's. Each D and B must be connected to exactly one A. The same rules also apply for all the child classes of C. If we have an instance of D3A, it must be connected to an object of type A. Because of the unconstrained multiplicity range, there can be an infinite number of different object combinations even, when there is only one A. One possible object diagram can be seen on Figure 2 B.

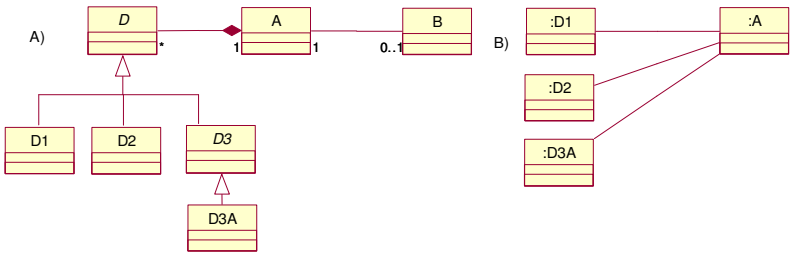


Fig. 2. Associations and multiplicity ranges

There are some features, like association classes that are not present in UML object diagrams. One way to present them in object diagrams is to have an object with a link to objects in both ends of the association, instead of creating an ordinary link between the two objects. Other exceptional situations are related, for instance, to complex constraints for the model, but we will not discuss them in this paper.

As the number of classes in a class diagram increases, it becomes more difficult to track all the possible choices in the instantiation process. Therefore, a guided instantiation process would certainly be beneficial.

3. Using Aspectual Patterns for Diagram Synthesis

In [Ham05], the concept of an aspectual pattern is proposed for specifying and representing various logically connected compositions of entities appearing in software artifacts. Aspectual patterns have been used, among other purposes, for guiding the development of a UML model for an application according to the specialization rules of a platform, when the rules are given in the form of aspectual patterns. Thus, aspectual

patterns have been used for producing UML models in a predefined way, providing a potential tool for our problem as well. In this paper, we apply aspectual patterns for producing object diagrams in a way allowed by a class diagram. We do this by transforming a class diagram into an aspectual pattern, and exploiting on the existing tool support for transforming an aspectual pattern into a task list for applying the pattern. In the sequel, we will refer to aspectual patterns simply as patterns.

Patterns have been used to represent heterogeneous collections of software entities, covering different artifact types and languages [Ham05]. Here we assume that a pattern is applied in the context of UML class and object diagrams. A pattern defines a set of *roles* that can be bound to model elements in the diagrams (like objects and links) and a set of *constraints* which defines the required structural relationships and properties of the model elements bound to the roles. In addition, a role has *multiplicity*, specifying how many model elements can be bound to the role when the pattern is applied: in an application of a pattern, a role can be instantiated as many times as allowed by its multiplicity, and each role instance is bound to exactly one model element.

Existing tool support [Hau05, Ham04] allows the specification of patterns and guides the user to apply the structural composition defined by the pattern in a model. In our case the pattern is a representation of a class diagram, and the user creates an instance diagram according to the pattern. The tool allows user interaction by generating tasks to be carried out by the user.

The idea is simple: each unbound role that can be bound in the current situation, taking into account the mutual dependencies of the roles and their multiplicities, becomes a task. The user can fulfill a task either by pointing out an existing element to be bound to the role, or by asking the tool to generate automatically a *default element* and to bind it to the role. The default element of a role is defined as a template with parameters whose values are taken from the elements bound to other roles, or provided by the user through a dialog. This mechanism turns patterns into a fairly powerful generative tool for various kinds of model synthesis.

As a trivial example of pattern-based model synthesis, assume we want to produce class diagrams that follow a simple rule: there should be one composite class which has an arbitrary number of other classes in part-of relationship with the class. This rule can be represented by a pattern with three roles: the composite class (say, Comp), the part classes (say, Part, with multiplicity "*"), and an aggregation relationship role. The latter obviously depends on its end class roles, Comp and Part. The pattern and a sample diagram produced using the pattern is shown in Figure 3.

In Figure 3, the roles are shown as small white circles, and an arc means that the source role depends on the target role. The multiplicities are associated with the roles as well. Note that we have introduced an additional dependency from Part to Comp; the only reason for this is to guarantee that the Comp class is bound before the Part classes: it seems logical to introduce first the parent and then its parts. The role instances are marked as grey circles besides the roles, each instance bound to a model element (dashed line).

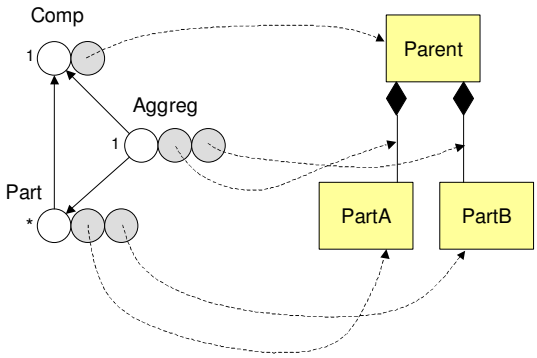


Fig. 3. Pattern-driven synthesis of UML diagrams.

The example pattern is used by the tool as follows. Since the only role not depending on other roles is Comp, the tool first generates a single task "Provide Comp". The user could now ask the tool to generate one, and the tool asks for the name in a dialog. After getting the name, the tool generates a corresponding class and binds it to the role (instance) Comp. Now the Part role becomes possible to bind, since there are no more unbound roles that it depends on. Thus, the tool generates task "Provide Part", and again asks for a name. This task is optional, because the lower limit for the number of Part classes is 0. After the user has given the name, the tool generates a Part class with the given name. As a result, a new task for binding the aggregation relationship role is generated. Since a name is not required for the relationship, the user can let the tool do this task automatically, if desired. In addition, there is a (optional) task for producing yet another Part class, because the multiplicity of Part allows an arbitrary number of those. In this way, the user can proceed as long as she wants. The task mechanism guarantees that the produced models conform to the given pattern.

Consider again the pattern in Figure 3. In the example, we synthesized a class diagram guided by the pattern. However, the pattern can be viewed as a representation of a class diagram with two classes and an aggregation relationship, with multiplicity * attached to the part class. If the generated classes are interpreted as objects and the generated aggregation relationships as links, the resulting diagram is actually an instance of the pattern class diagram. This illustrates our idea of using patterns as class diagram representations for producing instances of the class diagrams. In the following we show how the pattern can be constructed in a systematic way, given a class diagram.

4. Transforming Class Diagrams to Patterns

In this section, we describe the transformation from a class diagram to an aspectual pattern and different roles we use in patterns. We can use the guidelines of a class dia-

diagram instantiation described in Section 2 as the basis of the pattern construction process.

The pattern we are creating can be seen as a bridge between the model and the instance. In patterns, we can refer to the element of the class diagram and the object diagram we are creating. For example, there could be a role, which is bound to a class in a class diagram and another role, which represents an item of an object diagram. The conversion is done stepwise, moving from one class to another using the associations to navigate through the class diagram step by step. New roles are added to the pattern and connected to existing ones. The process stops when all the classes have been considered.

The pattern is used as an instantiation plan, where the user follows a given instantiation schema. The instantiation order follows roughly the order in which the pattern is created. The instantiation is started from the same point where the pattern creation is started.

Figure 4 depicts an example pattern specification and instantiation. On the left hand side we have the model to be instantiated, in the middle there is the pattern model for the instantiation and on right hand side we have the instance. We use the same pattern notation as in Figure 3. The pattern construction is started from class A and the first element added is the multiplicity role for class A.

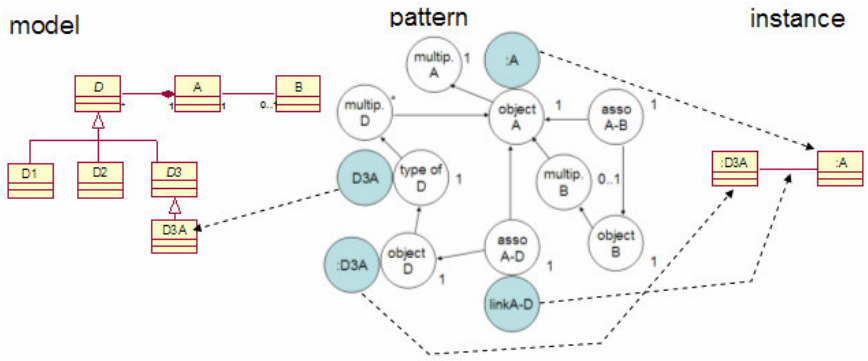


Fig. 4. Model, pattern, and instance and their relations

On the pattern level, there are roles for metamodel elements like attributes, classes, associations, and multiplicities. Roles for the classes can be *type roles* for choosing the type of the class from the model or *object roles* for providing an object to the instance. When a class diagram is instantiated, we can use already existing objects instead of creating new ones. Same object can play more than one role in the instance level. Type roles are needed if the class has subclasses. If a class has attributes, we need to have a method for setting a value to the attributes. For this we use *attribute roles*¸ one role for each attribute.

There can be several associations connected to one class, each with different multiplicities, thus we need a method to separate different multiplicity values. We cannot describe all the different multiplicities only by giving multiplicity values to type or object roles. Instead, we use *multiplicity roles* to describe the multiplicity values of

the class. The multiplicity roles have cardinality value, which is used to describe the multiplicity values of the corresponding association. Only multiplicity roles have varying cardinalities, all the other roles have cardinality one. The instances of associations (links between objects) also need to be described. We use *link roles* to describe the links.

The roles depend on each other and the dependencies and roles together form the pattern. The instantiation order is opposite to the dependency flow. The starting point is the first role added to the pattern from which there are no dependencies to any other role. All the other roles have dependencies to at least the starting point. New multiplicity roles depend on the object role of class in the other end of the association. The type roles depend on multiplicity roles. Object roles depend on either type roles, but if no type role is needed they depend on multiplicity roles. Links are made between objects, thus, the link roles depend on the object roles. Attribute roles depend on the object role they are related to.

In addition to classes and their relations, the model can have constraints. A systematic conversion of constraints to patterns is, however, beyond the scope of this paper. Thus the constraints are ignored in pattern construction process altogether.

In order to get the maximum amount of guidance, we need to have a good starting point for the process. A starting point is a central class in the model and it is decided by the designer. There are no foolproof methods for defining the starting point automatically. If we choose it randomly, it can limit the choices the user can make. For example, a randomly chosen starting point can lead the user to instantiate classes that would be optional if another instantiation order had been used.

To give an example, let us consider the class diagram in Figure 2. If we have class A as the starting point, the tasks for creating class B and C are optional. If instantiation is started from class B, we are committed to have it. The same thing happens if we take any type of class C first, we are then committed to have at least one instance of class C. There are, however, some guidelines that can be used to find out a good starting point. Association classes and child classes of a hierarchy should usually be avoided. The same rule holds for aggregated classes. They are not central elements and it is usually better to start from a more essential part of the system. Abstract classes can be chosen as a starting point although they themselves cannot be instantiated.

However, the best starting point is often related to the semantics of the concepts in the model, rather than to the structure of the model. What the diagram and its elements are representing is more important than the structure of the diagram. Thus it is better to leave the choice to the designer of the instantiation process. In addition to the starting point, the designer also needs to decide the cardinality value of the first element.

The pattern creation is started by adding object and type roles for the first class. After that, the associations connected to the class are used to navigate to the next classes. The associations back to the previous class need special attention. We deduct one from the multiplicity of all such associations. In case of multiplicity ranges, the deduction is made to both the lower and upper bound of the range. If the result is zero, the association is ignored. This is done because there already is a connection to one object of required type, namely the previous class. For each class encountered, we perform the following operations:

1. We check if the class already exists in the pattern. If the class already exists, we can move to the next phase. Otherwise, we check if the class has child classes and add a type role for the class if needed. After that, we need to check the subclasses of the class. For each subclass that has associations connected directly to it, we create a new object role. If none of the subclasses have their own associations, we create only one object role. The created object roles depend on the type role.
2. We add attribute roles for the newly created object roles and add a dependency between the attribute role and the object role.
3. We create a multiplicity role with cardinality of the multiplicity of the association that was used to get to the class. We connect the multiplicity role to the type role. If the class has no type role, we connect the multiplicity role to the object role.
4. We add link role to the pattern and connect it to the previous object role and the newly created multiplicity role.
5. The object roles created in phase 1 have a corresponding class in class diagram. We use the associations of these classes to navigate to other classes one by one. The associations of the parent classes are also used to find the connections.

This process is recursively repeated for each new class encountered. The process stops when there are no associations to navigate into or they all lead to classes that already exist in the pattern.

In Figure 4, we started the instantiation from class A. We add the necessary roles for the class and check the associations connected to it. There are two associations, one to class C and another to class B. Class C has child classes, and therefore we need a type role. The type role depends on the multiplicity role and the object role depends on the type role. As a result we get the pattern shown in Figure 4.

5. MADE Tool Platform

In order to demonstrate our task-driven approach to class diagram instantiations, we use a prototype tool environment known as MADE [Ham04]. The MADE platform itself is the result of integrating three different tools: JavaFrames [Hak01], xUMLi [Air02], and Rational Rose [Rose]. JavaFrames is a pattern-oriented development environment built on top of Eclipse [Eclipse]. Rational Rose is used as a UML editor. The third component, xUMLi, is a CASE tool-independent platform for processing UML models and is used for integrating JavaFrames and Rational Rose. There are newer and more sophisticated CASE-tools, but Rational Rose is still widely used by industrial partners related to MADE project. The tool has been developed to achieve a stepwise modeling and architecting development environment and has been used to manage different kinds of development scenarios [Ham05].

In this paper, we exploit the MADE tool concepts and features to assist users through class diagram instantiations. Currently MADE supports only class diagrams in UML, so we will use class diagrams to represent object diagrams, too. Accord-

ingly, we will use the role types MADE provides for class diagrams to represent patterns bound to object diagrams.

The specification for instantiating an arbitrary class diagram is given to the MADE tool in terms of a pattern-based tool concept, which we will refer to as MADE pattern. In the context of this work, a MADE pattern can be viewed as a configuration that captures a graph of nodes and edges. The nodes represent the objects and the edges represent the links between objects. To be able to define a pattern independently of any concrete class instantiation, a pattern is defined in terms of element roles rather than concrete objects and links; a pattern is instantiated in a particular instantiation context by binding the roles to concrete elements.

Pattern roles are attached with a number of properties. Each role may have a set of constraints. Constraints are structural conditions that must be satisfied by the concrete element bound to a role. A constraint of a UML association role P, for example, may require that the link bound to P must appear between the objects bound to certain UML class roles Q and R. A cardinality value is defined for each role. The cardinality of a role gives the lower and upper limits for the number of the elements bound to the role in the pattern. For example, if a UML class role has cardinality value 0..1, the corresponding instance is optional, because the lower limit is 0. Roles may depend on other roles. For example, there is a dependency from role P to role Q since the binding of P (creating a link) depends on the binding of Q and R (creating the objects). In this case, two objects should be bound to role Q and R before these two objects are used when binding role P to a link between the two objects.

For generating the objects and links, the MADE tool defines a default element for every role. If a role with a default element specification is to be bound during the pattern instantiation process, the binding can be carried out by first generating the default element according to the specification, and then binding the role to this element.

Figure 5 shows an overall view of the MADE environment. Rational Rose represents the upper part of the environment. In our example case, the object diagram is drawn as a class diagram in Rational Rose. (The Rational Rose Enterprise edition the MADE is incorporated to does not support object diagrams.) The model that is instantiated is on the left and the object diagram is on the right. The left view represents the part of the environment where patterns are specified and applied. Patterns are represented by circular graphical icons. In the MADE environment, patterns are instantiated as extensions of other patterns. For example, the pattern ‘bike_inst’ is an instantiation of pattern concrete_bike_inst, which specifies the instantiation steps required for the bike example presented in this paper. When a pattern is selected, MADE transforms the pattern into a task list. This is done by generating a task for each unbound role that can be bound in the current situation, taking to account the dependencies and cardinalities of roles. The task view displays the tasks implied by the pattern. This view is divided into two panes: task titles are shown in the upper pane and detailed task descriptions are presented in the lower one. In the example figure, a task for selecting a proper seat type is shown. The MADE tool incorporates wizards to help the user carrying out the instantiation process. We use the wizard to select the classes from the model.

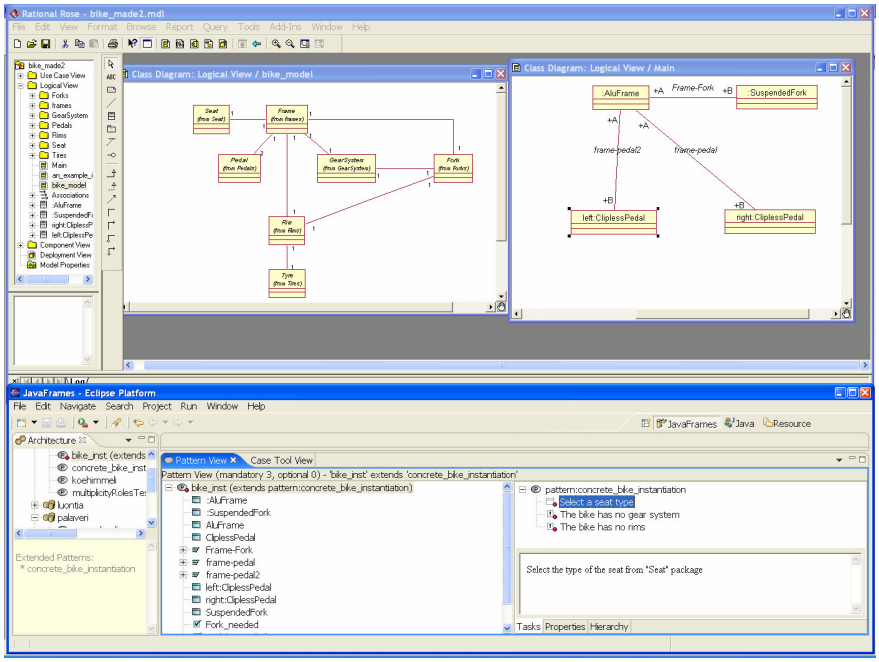


Fig. 5. MADE and Rational Rose used in an instantiation task.

There are two kinds of tasks: mandatory and optional. They both have their own graphical presentation; a hollow dot is used for the optional task and a filled red dot for the mandatory task. In the context of class diagram instantiation, executing a task means instantiating a class and adding the instance of that class to the object diagram. Consequently, a binding between a pattern role and the generated element is established and recorded by the tool. The middle view depicts the bindings that have been performed. In addition to the bound roles, the view shows the links that exist between the class instances.

As the object model can be freely edited through the UML editor, some bindings in an existing pattern become invalid or certain constraints defined by the pattern are violated. For example, certain objects might be accidentally deleted or the names and types of objects might be illegally modified. In this case, the pattern tool warns the developer about the inconsistencies and proposes corrective actions. It is then up to the developer to either correct the situation or ignore the warning.

6. An Example Instantiation Scenario Using MADE

As an application of class diagrams, consider the problem of product configuration. There are a wide range of products that are assembled from components ranging from

personal computers to trucks. Some of the components may be optional while others are necessary to get a complete product. The different components need to be assembled together before the product is complete. The product possible configurations can be given using UML class diagrams. The configuration models can also be used to help the assembly by defining order in which the parts of a product need to be assembled. Using a task-driven approach, we can also give the user the tasks in the right order.

Our example product, which is a mountain bike, consists of over 15 different kinds of components including frame, fork, brakes, shifting system, springs, rims etc. Each component (e.g. frame) can be changed, thus leading to a new configuration. The bike factory can design a range of predefined configurations as their market models. In addition to the models given by the factory, the customer could also create a custom bike by choosing her own combination from various parts. Some of the parts like reflectors, cat-eyes, and mudguards are optional while others are compulsory. Instead of giving a customer full control over all the different components, the bike manufacturer could provide a set of component collections the user can choose from.

We have a simplified model of a bike as our example case. A class diagram illustrating the configuration of the component is given in Figure 6. To keep the model simple, the component ranges are presented in their own packages and only base classes are used in the configuration model. An example component hierarchy is presented in Figure 7.

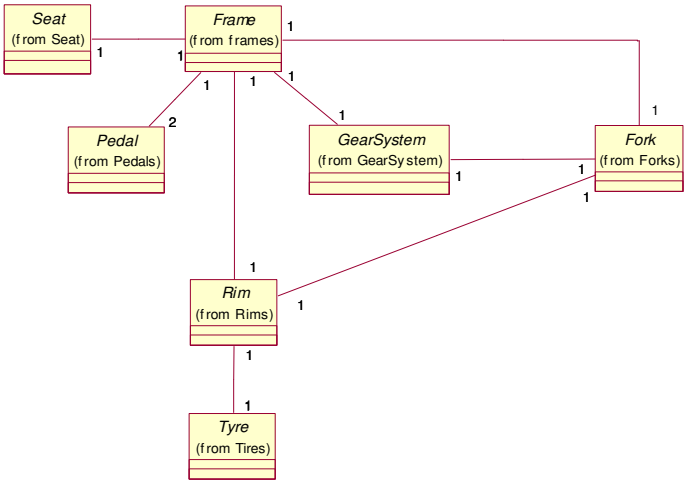


Fig. 6. A (simplified) configuration model of a mountain bike.

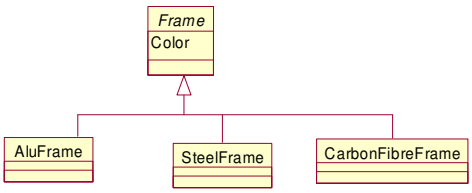


Fig. 7. Different frame types.

The bike model is used in the creation of an instantiation pattern. The MADE environment generates the tasks that the user needs to complete in order to get a legal bike configuration based on given pattern. The tool allows designer to add additional instructions to the roles giving the user hints on e.g. how to complete tasks. The constraints were ignored on pattern creation but the designer can add them to the MADE pattern by hand. In addition to the constraints found in the model, the designer can also introduce additional constraints. Using the constraints, we could, for example, force the user to have rims of rear and front wheel to be of the same type.

The pattern model in the MADE tool is very similar to the one presented in Section 4, but there are some limitations and differences. In this work, we use four different role types: UML class roles, UML association roles, attribute roles and informal roles; the purpose of the latter is to specify arbitrary informal tasks. UML class roles are used to present type and object roles, UML association roles are used for links between objects, and informal roles are also used to represent the multiplicity roles.

The pattern derived from the bike model is shown in Figure 8. The association roles were left out from the pattern to clarify the figure by limiting the amount of visible roles and dependencies. Roles without an explicit multiplicity value have a multiplicity of one. A frame was chosen as the starting point of the pattern construction process. A pending instantiation process where this pattern is used can be seen in Figure 5.

The MADE environment defines only four different multiplicity values (1, 0..1, *, and 1..*) instead of free range of multiplicities, thus we need to have a workaround to present for example constant multiplicity values like 2. If a fixed multiplicity is greater than 1 but not * we have same number of roles with cardinality value of one. In our example case, the bike has two pedals. We cannot use a role with a cardinality of 2, therefore, we need two multiplicity roles with cardinality of 1. To present multiplicity ranges like 2..4, we can use roles with cardinalities of 1 and 0..1 in combination.

The MADE tool does not accept circular relations in the patterns. Because circular relations are banned, the tool cannot be used to present patterns with recursive loop structures. We cannot use the pattern model described in Section 4 without small changes to the pattern creation process. We simplify the process by ignoring the associations back to the previous class completely. If there are associations with lower bound of multiplicity greater than one in both ends of the association in the model, the simplified pattern may produce invalid instances. Such associations are rare, but when encountered, the designer of the instantiation process needs to device a workaround to correct the pattern. Restriction on loops also leaves structures like the Composite de-

sign pattern [GOF, p. 163] out of the scope of our straightforward MADE instantiation.

If circular relations are supported, there are models that can produce infinite number of objects. Figure 9A shows a simple class diagram with two classes. The user can create a small but complete object diagram presented in Figure 9B or continue to create new elements indefinitely (Figure 9C).

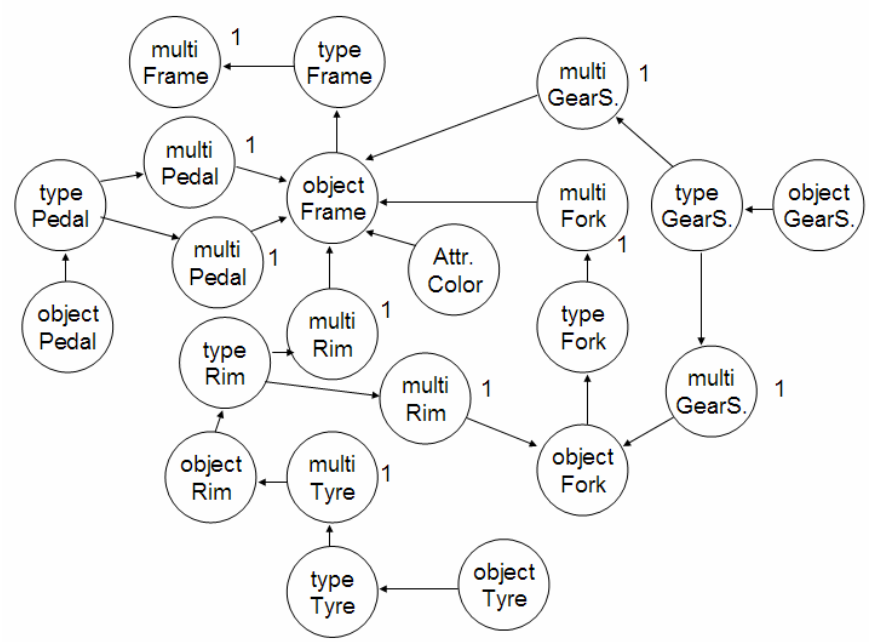


Fig. 8. A pattern model of a bike

When the pattern presented in the Figure 8 is applied in the MADE environment, the first task presented to the user is the selection of a frame type. After that, the user needs to provide an instance of the frame. After creation of the frame, the user can continue by choosing seat, pedals, or fork. New tasks appear after the old ones are completed. The links between the objects are created semi-automatically. The user creates the links by acknowledging the creation. An example object diagram is shown in Figure 10.

The task-driven process can also be used to verify and complete existing object diagrams. Instead of creating new elements to the object diagram, the user binds roles to already existing objects. After the binding are done, the MADE tool can be used to complete the diagram and verify the completeness of the diagram.

Having only one starting point can be seen as an undesired restriction, the freedom of choice can be increased by introducing several starting points. This can be done by creating multiple patterns, each with a different alternative starting point. Depending on the chosen starting point, the appropriate pattern will then be activated.

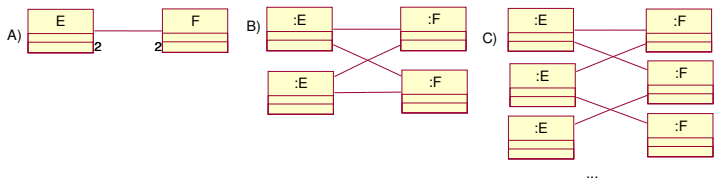


Fig. 9. Potentially endless instantiation of a simple model

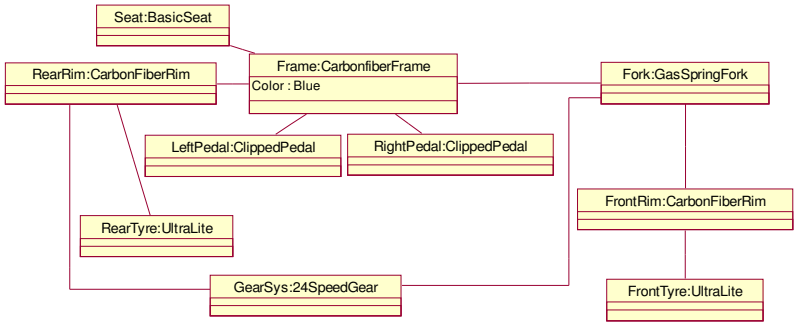


Fig. 10. An instance of the bike model

7. Related Work

The problem of creating legal models based on a given metamodel is addressed, e.g. by all UML CASE-tool vendors, in their products, like Rational Rose[ROSE], Together [TOGETHER], and ArgoUML [ARGO]. In these tools, changes in the metamodel require changes in the tools themselves. Hence, these tools are not optimized for instantiation of arbitrary class diagrams.

This problem is solved, for instance, in so-called meta-case tools, such as MetaEdit+ [ME01, ME05], ObjectMarker [MV], GME (Generic Modeling Environment) [GME], and The Coral Modeling Framework [Coral]. Typically, meta-case tools offer a metamodeling tool suite (i.e. framework) for defining modeling concepts, their properties, associated rules, and symbols needed to specify, implement, and use modeling languages.

As stated, basically all the UML model editing tools apply “syntax-directed editing” in the sense that they allow the user to create only models which comply with the UML metamodel, or at least aim to do so. However, in these tools the users have to know the underlying metamodel. If the user, for instance, tries to instantiate a wrong type of element in a situation, the tools typically just deny the action, instead of guiding in the instantiation. Therefore, the main advantage of our approach is that the

task-driven process genuinely guides the user, and relieves her from knowing the underlying metamodel.

There are also other task-based tools, like various workflow tools, e.g. Metastorm [Metastorm] and Telelogic Popkin [Popkin], but they do not address the problems presented in this work. Typically, they are not aware of the underlying models in any way.

8. Discussion

In this paper, we showed how class diagrams can be instantiated in systematic task-driven way. We showed how to make a conversion from a class diagram to a pattern model for instantiation and presented an implementation based on the MADE platform. Our early experiences with the task-driven instantiation suggest that the approach can be used to facilitate the instantiation process. The instances created using our approach can be verified as they are constructed and their completeness can be assured. Even with little knowledge of modeling languages, the user can create instances of a model using this approach. However, our approach needs to be thoroughly evaluated against real world examples, which may require the instantiation of larger complex models.

Using a combination of the MADE environment and Rose can be difficult for a person with little experience on the modeling tools. A simpler user interface should be provided if the task-driven approach is to be used with models that are not related to software. Usability tests should be performed in order to verify the applicability of our approach using real world examples.

Acknowledgements

This research has been financially supported by the National Technology Agency of Finland (Project Inari), Nokia, Plenware Group, TietoEnator, Plustech, and Geracap.

References

[Air02] Airaksinen J., Koskimies K., Koskinen J., Peltonen J., Selonen P., Siikarla M., Systä T.: xUMLi: Towards a Tool-independent UML Processing Platform. In: Proceedings of the Nordic Workshop on Software Development Tools and Techniques (K. Osterbye, ed.), NWPER 2002, IT University of Copenhagen, 2002.

[ARGO] ArgoUML, http://argouml.tigris.org/, 2005.

[Coral] Marcus Alanen and Ivan Porres: The Coral Modelling Framework. In Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004, Jul 2004.

[Eclipse] Eclipse, Eclipse website, http://www.eclipse.org, 2005.

[GME] GME: The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/, Institute For Software Integrated Systems, 2005.

[GOF] Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison Wesley, 22nd printing, July 2001

[Hak01] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Generating application development environments for Java frameworks. Proc. GCSE 2001 (3rd International Conference on Generative and Component-Based Software Engineering), September 2001, Erfurt. Lecture Notes in Computer Science 2186, Springer, 163-176.

[Ham04] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen. Adaptable Concern-based Framework Specialization in UML. *In Proc. ASE 2004*, pages 78–87, Linz, Austria, September 2004.

[Ham05] Hammouda I.: A Tool Infrastructure for Model-Driven Development Using Aspectual Patterns. *In Sami Beydeda, Matthias Book, and Volker Gruhn, eds., Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Hau05] Hautamäki J.: Pattern-Based Tool Support for Frameworks – Towards Architecture-Oriented Software Development Environment. PhD thesis, Tampere University of Technology, Publication 521, 2005.

[ME01] MetaCase Consulting, Domain-Specific Modeling: 10 Times Faster than UML, MetaCase Consulting, Inc., Jyväskylä.

[ME05] MetaCase Consulting, MetaCase Consulting website, http://www.metacase.com, 2005.

[Metastorm] Metastrom, Metastorm Inc. webpage, http://www.metastorm.com/, 2005.

[MV] MarkV Homepage, http://www.markv.com/, MarkV Systems, 2002.

[OMG] Object Management Group, OMG Unified Modeling Environment Specification, Version 1.5, May 2003.

[Popkin] Telelogic Popkin, http://www.popkin.com/customers/customer_service_center/enterprise_architecture_resource_center/bpm.htm, 2005

[ROSE] Rational Rose, http://www-306.ibm.com/software/rational/, IBM, 2005.

[TOGETHER] Together, http://www.borland.com/us/products/together/index.html, Borland, 2005.

Practical Refactoring of Executable UML Models

Łukasz Dobrzański, Ludwik Kuźniarz

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
PO Box 520
S-372 25 Ronneby, Sweden
ludo04@student.bth.se, lku@bth.se

Abstract. One of the inevitable negative effects of software evolution is design erosion. Refactoring is a technique that aims at counteracting this phenomenon by successively improving design of software without changing its observable behaviour. This paper presents an overview of recent approaches to UML model refactoring, followed by results of an initial study on refactoring of executable UML models, i.e. models that are detailed enough to be automatically compiled to executable applications. It focuses on identification of refactoring areas in models built in Telelogic TAU, and it contains a description of application of several different refactorings to an exemplary executable model.

1 Introduction

Software maintenance is one of the key issues in the overall software construction and management. The issue relates to the fact that software systems live and evolve in time. Chapin *et al.* [3] define *software maintenance* as "the deliberate application of activities and processes (…) to existing software that modify either the way the software directs hardware of the system, or the way the system (…) contributes to the business of the system's stakeholders." In the modern approach, software maintenance encompasses activities and processes involving existing software not only after its delivery but also during its development. Worth mentioning is the fact that nowadays more than 80% of total software life-cycle costs is devoted to its maintenance [13].

Chapin *et al.* [3] propose a semi-hierarchical classification of the types of software maintenance that bases on "objective evidence of maintainer's activities." Their categorisation groups twelve types of software maintenance into four clusters, which are gathered in a decision tree shown in Figure 1.

According to Bennett & Rajlich [1], currently there is no one commonly accepted definition of *software evolution*, and in a wide sense, the term is often used as a synonym of software maintenance. However, Chapin *et al.* [3] distinguish between software maintenance and software evolution. In their opinion, the latter occurs when *enhancive*, *corrective*, *reductive*, *adaptive* or *performance* maintenance is carried out.

In other words, software evolution happens when business rules, or software properties that are sensible for customer are changed.

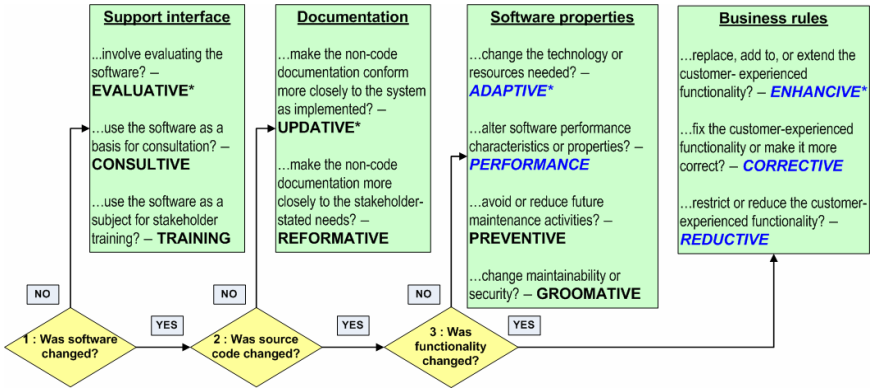


Fig. 1. Software maintenance classification [3]. The tree is read from left to right, and in each cluster – from bottom to top. Italic font indicates that the type of maintenance causes evolution. * indicates the default type in a cluster. "…" means *"Did the maintenance activities…"*

As indicated by Van Gurp & Bosch [21], despite many years of research and many suggested approaches, it is inevitable that a software system finally erodes under pressure of the ever-changing requirements. This negative effect of software evolution is known in the literature as *software aging* [11], and one of its dimensions is *design erosion*. During evolution, almost each change of requirements posed on a software system enforces introduction of small adaptations to its design. These adjustments are taken in the context of (1) all previous changes, and (2) predictions about possible future changes that may need to be made. It is obvious that some of these predictions can be wrong. In a consequence, the system may evolve in a direction where it is hard to make necessary adjustments. Two trends that were observed by Van Gurp & Bosch [21] are that (1) fixing design erosion is expensive, and (2) eroded software may become an obstacle for further development.

Refactoring[1] is one of the techniques that aim at counteracting the phenomenon of design erosion. It relies on successively improving design of software without changing its observable behaviour. In the context of the categorisation provided by Chapin *et al.* [3], refactoring is an activity that directly supports the types of maintenance present in the software properties cluster. It is particularly suitable for *groomative* maintenance that involves among others "replacing components or algorithms with more elegant ones, (…) changing data naming conventions, altering code readability or understandability [ibid.]." Refactoring can be also useful in *preventive* maintenance activities, which example is "participation in software reuse [ibid.]." *Adaptive* maintenance encompasses activities like "reallocating functions among components or subsystems, (…) changing design and implementation practices [ibid.]", what also can be achieved by the use of refactoring.

[1] For an extensive literature survey on software refactoring, performed in the frames of "The Refactoring Project", refer to [8].

The remainder of the paper is organised as follows. Section 2 provides an overview of recent approaches to refactoring of UML models, Section 3 introduces the notion of executable UML, Section 4 concerns refactoring of TAU executable UML models, focusing on identification of refactoring areas, and Section 5 presents several refactorings applied to an exemplary TAU model.

2 UML Model Refactoring

Refactoring was primarily used in the context of code and it aimed at improving its quality characteristics. Several catalogues of refactorings were published, starting with the most commonly known and used one introduced in the Fowler's book [5]. Although the book concerns code refactorings, over 60% of them are illustrated with the use of UML class diagrams. This observation motivates a question whether code refactorings can be applied to UML models. Zhang *et al.* [21] states that it is obvious that some code refactorings can also be used to transform class diagrams. According to Boger e*t al.* [2], for some refactorings, like e.g. *Extract Method*, it is natural to apply them on the code representation level. Other, like *Rename Class* or *Pull Up Method* can be applied both on the code as well as on the model level, and refactorings like *Replace Inheritance with Delegation* or *Extract Interface* are more apparent on a model level.

According to France & Bieman [6], applying refactorings on an abstract view of the system facilitates meeting design goals and addressing deficiencies uncovered by evaluations, i.e. improving specific quality attributes directly on a model. It also enables to explore relatively cheaply alternative decision paths in system's design. Sunyé *et al.* [17] mention that the primary advantage of UML over other modelling languages, in the context of model refactoring, is the syntax, which is precisely defined by a metamodel. Therefore, the metamodel can be used to control the impact of a transformation and provide means for ensuring its behaviour-preservation.

There are several attempts to perform refactoring on models expressed in UML. Most representative of them are briefed in the sequel.

2.1 UMLAUT

Sunyé *et al.* [17] attempted to transpose some of Robert's [16] refactorings to UML models. In the result of their work, an initial set of UML class diagram refactorings has been created. For each refactoring, a textual description of preconditions that have to be satisfied before performing the transformation is provided. Besides class refactorings, Sunyé *et al.* [17] describe six novel statechart refactorings. Behaviour-preservation of each statechart refactoring is expressed with the use of pre- and postconditions specified in OCL at the metamodel level. For the sake of simplicity, no details of how each refactoring accomplishes its intent, is given. However, the authors suggest the use of their UML general-purpose transformation framework called UMLAUT (Unified Modeling Language All pUrposes Transformer).

2.2 Refactoring Browser for UML

Apart of considering refactorings of class diagrams, Boger *et al*. [2] focus on activity diagram and statechart refactorings. Some refactorings of class diagrams and all of activity and statechart diagrams described in their paper were implemented in a refactoring browser for UML as a part of the Gentleware/Poseidon for UML tool. Behaviour-preservation of each refactoring is defined in a form of preconditions that are evaluated for currently selected model elements. Each precondition is mapped to appropriate messages, which are presented to the user in the case of its violation. These messages correspond to conflicts that are grouped into warnings, indicating that the refactoring might cause side effects, while leaving the model in a well-formed state, and errors, indicating that the refactoring will break the consistency of the model.

2.3 SMW toolkit

Porres describes [15] how UML model refactoring can be implemented as a sequence of transformation rules or guarded actions. Each transformation rule consists of five elements: its name, a documentation string, a sequence of formal parameters, a guard defining when the rule can be applied, and a body, i.e. the implementation of the rule. A rule takes one or more model elements as actual parameters and performs a basic transformation action based on these parameters.

Porres presents an execution algorithm for the transformation rules and describes a mechanism that ensures that the transformed models are well formed. However, he does not discuss the behaviour-preservation property of refactorings.

In the absence of a standardized language for model transformations, Porres implements refactorings using SMW (Software Modeling Workbench) – a scripting language based on the Python programming language. In many respects, SMW is similar to OCL, but it additionally provides a set of operations enabling implementation of model transformations. The idea of extending OCL with action features has been already discussed by Pollet *et al.* [14]. The main advantage of this approach is the possibility of implementing model transformations in one language along with defining their pre- and post-conditions.

Models can be accessed from SMW scripts via a metamodel-based interface. Each metaclass from the metamodel is represented in SMW as a Python class and each element in a model – as an instance of an appropriate class. The classes representing the metamodel have the names, attributes and associations as defined in the UML 1.4 standard [9].

In order to validate the execution algorithms and to evaluate how difficult it is to implement new refactorings in practice, Porres constructed – using the SMW toolkit – an experimental, metamodel-driven refactoring tool, and integrated it with an existing UML editor.

2.4 C-SAW & GME

Zhang *et al.* [21] describe an approach to model refactoring with the use of the Constraint-Specification Aspect Weaver (C-SAW) model transformation engine, a plug-in component for Generic Modeling Environment (GME). GME is a UML-based metamodelling environment that can be configured and adapted from loaded into it meta-level specifications (called the *modelling paradigm*) that define all the modelling elements and valid relationships between them in a particular domain. The UML/OCL meta-metamodel of GME is based on its own specification instead of Meta-Object Facility (MOF). However, an ongoing project incorporates OMG's MOF into GME.

A prototype model refactoring browser operating with the underlying C-SAW has been developed as a plug-in for GME. It provides automation of generic pre-defined refactorings within the GME metamodel domain. Additionally, it enables the specification of user-defined refactorings of both generic and domain-specific models (e.g. Petri Nets, AQML models, or finite state machines).

Refactoring strategies for user-defined refactorings can be specified and implemented using a special underlying language, called Embedded Constraint Language (ECL). Users of the refactoring browser are also allowed to customize pre-defined refactorings by modifying the corresponding ECL code. Generally, a refactoring is composed of a name, several parameters, preconditions and a sequence of strategies.

According to Zhang *et al.* [ibid.], ECL is an extension of OCL providing many of the common features of OCL, such as arithmetic, logical and collection operators. Additionally, it provides special operators supporting model aggregates, connections and transformations (e.g. addModel, setAttribute or removeNode) that can access model elements stored in GME.

2.5 *GrammyUML* – Source-Consistent UML Model Refactoring

Van Gorp *et al.* [19] point out that, although model refactorings are expressed at the design level, they must be aware of all the detailed code-level issues. This problem was already noticed by Demeyer *et al.* in a research paper concerning UML shortcomings for coping with round-trip engineering [4], as well as by Sunyé *et al.* [17] who provide two examples of refactorings (*Move Method* and *Specialization*) which pre- and postconditions – in the absence of information about method bodies of particular operations – cannot be verified at the model level.

Van Gorp *et al.* [19] argue that the UML 1.4 [9] metamodel is inadequate for maintaining the consistency between design models and corresponding program code. The UML 1.4 metamodel considers method bodies as implementation specific and therefore, typical UML tools treat them as "protected areas", which must be supplied manually and are not altered during code (re)generation. After refactoring a UML model and next regenerating a source code, it is common that inconsistencies appear in these "protected areas". For example, in the case of *Pull Up Method* refactoring, a UML 1.4 metamodel based tool must be able to decide on textual identity of methods, in order to remove from superclasses all copies of a pulled up method. Even in the

case of the simple *Rename Class* refactoring, such a tool is not able to update the refactored class's name in type declarations, type casts and exceptions. On the other hand, given a precise model of statements in a method body, a UML tool would be able to perform even such typical code level refactorings, like *Extract Method*.

In order to prove that the UML 1.4 metamodel is almost sufficient to allow for expressing source-consistent model refactorings[2], Van Gorp *et al.* [19] carried out an experiment, which goal was to provide concrete suggestions on realization of an ultimate UML refactoring extension. Within the framework of the experiment, they constructed *GrammyUML* metamodel, which bases on the UML 1.4 metamodel and includes eight additive extensions that allow for, among others, modelling statements in method bodies and use of typed local variables in a given scope. With the purpose of verifying access-, call- and update-preservation of refactorings, several stereotypes have been defined and incorporated into *GrammyUML*, along with four new refactoring Well-Formedness Rules.

In the next step, Van Gorp *et al.* [19] described in OCL, at the level of *GrammyUML* metamodel, the *refactoring contracts*, i.e. associated bad smells, pre- and postconditions, of two sample refactorings, namely *Extract Method* and *Pull Up Method*. As already stated, it would be impossible to express refactoring contracts of these refactorings at the level of UML 1.4 metamodel.

Van Gorp *et al.* [18,20] validated their approach by implementing *Pull Up Method* refactoring in an open source UML CASE tool called Fujaba, with the use of Story Driven Modeling (SDM), a visual programming language based on UML and graph rewriting. The most straightforward solution would be to use instead of Fujaba an OCL-enabled tool, however they did not do this due to "the practical unpopularity of OCL (both in use by developers as in tool support) [18]."

In order to ensure appropriate source code regeneration from refactored models, Van Gorp *et al.* suggest introduction of a new component called *Code Preserver* into the Fujaba architecture. They define it as "a development tool component that stores all the required source code files from which a model is extracted in such a way that the complete system can be regenerated from a transformation of the input model [20]." The need for *Code Preserver* results mainly from the fact that *GrammyUML* metamodel, since it includes only a minimal set of information sufficient for reasoning about refactoring, does not contain all syntactically possible source code constructs.

3 Executable UML

Executable UML is considered to be a major innovation in the field of software development. According to Pender, the term executable UML is used to describe "the application of a UML profile in the context of a method that aims to automatically generate an executable application from an abstract UML model [12]." It is a graphical specification language, which combines a streamlined, computationally complete subset of the UML with execution semantics and timing rules. In contrast to traditional specifications, an executable specification can be run, tested, debugged and

[2] UML refactorings that maintain consistency between refactored model and underlying code.

measured for quality attributes. However, the main benefit of this approach is the possibility of fully automated translation of executable UML models into source code. Executable models confer independence from software platforms, what makes them portable across multiple development and execution environments.

There exist several both commercial as well as research attempts to achieve executable version of UML that differ in (1) means used to specify models, and (2) the way models are executed. In this paper, we consider executable UML models built in Telelogic TAU UML CASE tool. The behaviour of a TAU UML model and its implementation may be verified with the use of the *Model Verifier*. First, the *Application Builder* generates an executable program in the C language from the model linked with a predefined run-time library customized for simulation purposes. Next, the program is executed – either automatically or manually, i.e. in a step-by-step manner using various commands and breakpoints. The execution of the session can be traced graphically in state machine and sequence diagrams or textually in the output console window. If the application communicates with the environment, this also may be simulated by sending manually prepared signals.

In order to be executable with the *Model Verifier*, a UML model must have a certain level of completion. It must be composed of:

- a package (optional),
- a class diagram (optional),
- at least one active class (the so-called top-level active class),
- at least one state machine with an implementation (optional – in the sense that it can be implicit).

An active class, i.e. a class with its own thread of control, must have a port to be able to communicate with other active classes and/or its environment with the use of signals. A port – a named interaction point of an active class – is defined by the signals, usually encapsulated in interfaces, which it can transmit. A model with (internal or external) communication has:

- at least one signal,
- at least one port,
- an interface (optional).

The life cycle of an active class is described with a state machine named *initialize* or having the same name as its owner. The implementation of a state machine is visualized on a statechart diagram (alternatively – in a text diagram). Each active class may have its internal run-time structure defined in terms of other active classes, referred to as parts. A composite structure diagram may be used to both visualize this architecture as well as to express the communication within an active class by showing connectors between the ports of the parts. A special kind of ports, namely behaviour ports, may be used to enable the communication between a part and the state machine of an instance of the class that owns the part.

4 Executable UML Model Refactoring

All previous studies presented in Section 3 concern refactoring of non-executable UML models. The main difference between transformations used in these approaches

and refactorings of *executable* models relies on the fact that the latter ones have to take into account and update not only structural but also behavioural aspects of transformed models. The main challenges in the area of refactoring of executable models that base on UML 2.0 [10] result mainly from the necessity to consider following new features of the language: (1) cross integration of structure and behaviour, (2) support for component-based development via composite structures, and (3) integration of action semantics with behavioural constructs. With the intention of enabling a systematic approach to the mentioned issues, we introduce the concept of *trigger-elements* and *refactoring areas*.

Each refactoring has an associated *trigger-element*. From the practical point of view, a trigger-element of a refactoring is the type of a code/model element on which it can be triggered in an IDE/CASE tool. For instance, a trigger-element of *Inline Temp*, *Replace Temp with Query*, *Split Temporary Variable*, and *Remove Control Flag* is Temporary Variable. These are the refactorings, which one expects to be able to apply after selecting a Temporary Variable in the body of an operation belonging to a class, while browsing a model/code. It is noteworthy that there is difference between "triggerness" and "driveness" – refactorings are *triggered on* model/code elements and *driven by* the presence of bad smells.

All code trigger-elements can be classified into two disjoint groups: (1) *structural elements* and (2) *behavioural elements*. Method Body and its internals, i.e. Code Fragment, Literal Number, and Temporary Variable, belong to the latter group, and the rest of them are structural ones. Refactorings triggered on structural elements are *structure-triggered (S-T)*, and the ones triggered on behavioural elements – *behaviour-triggered (B-T)*. This implies that in programs written in object-oriented programming languages there are two *refactoring areas*, namely (1) structure – inter-related classes and their features, and (2) behaviour – bodies of methods.

In the context of UML, a refactoring area may be defined as a certain part of a model containing particular trigger-elements. Refactorings applicable for UML 1.x models are only the structure-triggered ones, because in these models there is only one refactoring area – the structure. On the other hand, in TAU executable models, six refactoring areas can be distinguished (see Fig. 2).

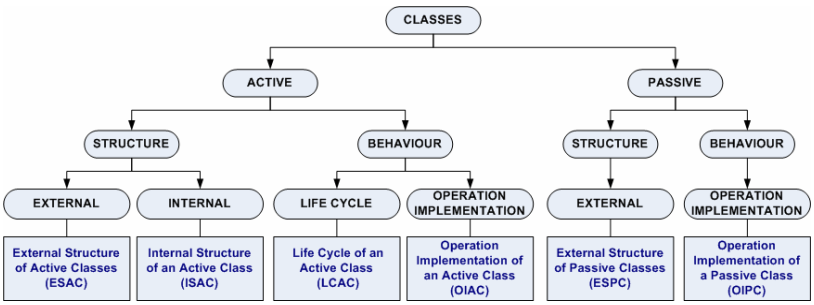


Fig. 2. Six refactoring areas in TAU executable UML models

Area 1 – External Structure of Active Classes (ESAC). The area consists mainly of active classes that have attributes, operations, and ports that require and realize single signals and/or whole interfaces. Additionally, active classes may have

(composite/shared) associations to passive classes and composite associations to their parts being other active classes. All these elements may be show in class diagrams, and some of them (e.g. parts and ports) additionally in composite structure diagrams. Candidates for transformations that can be triggered on model elements from this area are all of the structure-triggered code refactorings. However, in many cases their practical realization may considerably differ from the ones provided by Fowler [5]. Moreover, in the area there are potentially many other, so far unidentified refactorings triggered on among others ports, signals, timers, and interfaces.

Area 2 – Internal Structure of an Active Class (ISAC). The area consists of active classes that are parts of their container, and which communicate with each other by sending signals and invoking operations via ports wired by connectors. Some aspects of this area may be illustrated in class diagrams, but e.g. connectors – only in composite structure diagrams. As composite structures are new to UML 2.0, so far, there exists no literature concerning refactorings applicable to this area. Nevertheless, these transformations would deal mainly with reorganization of internal structure and communication infrastructure of active classes, and thus they have no equivalents among code refactorings.

Area 3 – Life Cycle of an Active Class (LCAC). The area constitutes implementation of a default state machine of an active class, represented in a statechart diagram. In this area, there are two kinds of trigger-elements: (1) (elements of) compound actions on transitions, and (2) states and transitions between them. Refactorings triggered on the elements from the first group are mainly specific versions of some behaviour-triggered code refactorings. Other, but already UML specific, transformations identified and described by Sunyé *et al.* [17] and Boger *et al.* [2] can be applied to the elements from the second group.

Area 4 – Operation Implementation of an Active Class (OIAC). The area constitutes implementation of an operation belonging to an active class. In the context of active classes, this implementation may be either state or stateless. However, the former solution introduces only new presentation elements for corresponding triggers being the same model elements, namely (elements of) various actions, as in the latter case. Refactorings triggered on elements from this area may be derived from code behaviour-triggered ones. However, their realizations may differ from the ones provided by Fowler [5] due to the possibility of among others communication with the use of signals and via connectors.

Area 5 – External Structure of Passive Classes (ESPC). The area consists of passive classes that may have attributes and operations as well as (composite/shared) associations and generalizations to other passive ones. All these elements may be show in class diagrams. Candidates for transformations that can be triggered on model elements from this area are the same as in the case of ESAC, i.e. all of the structure-triggered code refactorings. However, their practical realizations are usually simplified with respect to their equivalents from ESAC.

Area 6 – Operation Implementation of a Passive Class (OIPC). The area constitutes implementation of an operation belonging to a passive class. In the context of passive classes, this implementation may be only stateless, i.e. in the form of actions written in *U2 Action Language* contained in a text diagram. As in the case of OIAC, the refactorings triggered on elements from this area may be derived from

code behaviour-triggered ones, but their practical realizations are usually simplified with respect to their equivalents from OIAC.

5 Exemplary Refactorings in Practice

To facilitate the understanding of refactoring areas, we choose four transformations[3] and show how they can be applied to an exemplary TAU executable UML model of a satellite[4]. The selected transformations are:

1. Area ISAC – *Extract Port*[5] – triggered on a port of an active class, which is used for communication with several different parts. It relies on creating a new port and reconnecting some connectors of the old one to the new one.
2. Area LCAC – *Group States* [17] – triggered on a simple state in a default state machine of an active class. It relies on transforming the state into a composite one, and thus reduces the number of redundant transitions.
3. Area OIAC – *Replace Method with Method Object* [5] – triggered on an implementation of an operation of an active class, which is too long and cannot be decomposed with the use of other refactorings. It relies on turning the operation into a class.
4. Area ESPC – *Hide Delegate* [5] – triggered on a passive class. It relies on encapsulating it from other ones, and thus reduces the coupling between classes in the model.

5.1 Area ISAC – *Extract Port*. The top-level active class of the model is *Satellite*, which has several parts typed by *EarthCommunicator*, *Navigator*, and *InstrumentsController*. As can be observed in Figure 3, *pOutput* port of *earthCommunicator* serves for communication with two different parts with the use of two semantically unrelated signals – *plan*, containing the most recent plan of the mission, and *command*, carrying new instructions for scientific and navigational instruments. A refactoring that should be triggered on *pOutput* is *Extract Port*.

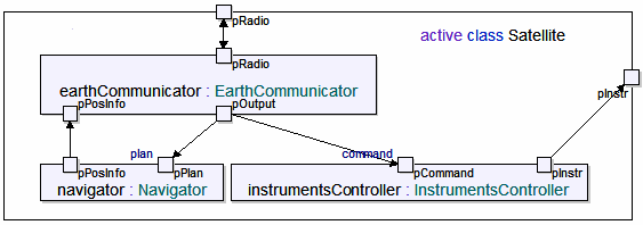


Fig. 3. A simplified composite structure diagram of *Satellite*

[3] It is worthy noting that refactorings *Replace Method with Method Object* and *Hide Delegate* can be triggered also on the elements in the OIPC and ESAC areas, respectively.

[4] For the sake of conciseness, only these parts of the model, which are important in the context of particular transformations, are presented.

[5] This refactoring has not been previously mentioned in the literature.

As the result of the refactoring, a new port *pCommand* is added to *EarthCommunicator*, and the connector that transmits *command* is reconnected to it. Next, on a class diagram showing *EarthCommunicator*, *command* is moved from to the list of signals required by *pOutput* to the one belonging to *pCommand*. Assuming that in all output actions – be it in a default state machine of *EarthCommunicator* or in bodies of its operations – signals are sent without *via* keyword, the refactoring finishes, otherwise each expression "*[output] command(instr) via pOutput*" has to be changed to "*[output] command(instr) via pCommand*". Subsequently, one can apply *Rename Port* refactoring to *pOutput* in order to give it a more meaningful name, e.g. *pPlan* (see Fig. 4). Finally, one can consider merging *pPosInfo* and *pPlan* in both *EarthCommunicator* and *Navigator* with the use of *Merge Ports* refactoring.

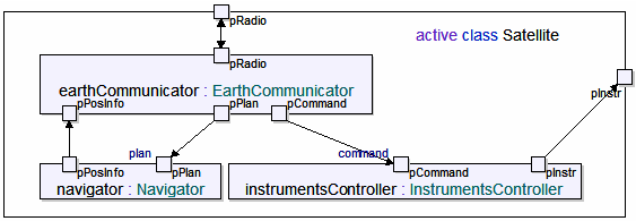


Fig. 4. A simplified composite structure diagram of *Satellite* after triggering *Extract Port* and *Rename Port* on *pOutput* port

5.2 Area LCAC – *Group States*. The lifecycle of *InstrumentsController* is defined by a state machine shown in Figure 5. Just after creation, an instance of the class finds itself in *Idle* state, in which it awaits for *command* signal sent by *earthCommunicator*. The signal triggers a transition to *Decoding* state. Next, after going through *Calculating* and *Encoding* states, the state machine reaches *Adjusting* state, in which it adjusts every 10 ms the instruments, as long as new instructions appear.

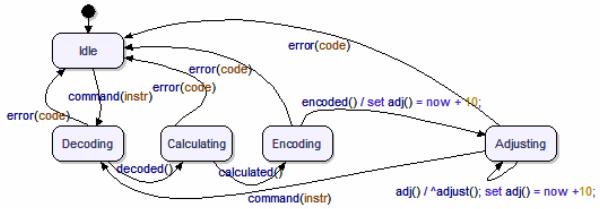


Fig. 5. A state chart diagram showing implementation of a default state machine of *InstrumentsController*

It can be noted that from each state there is a transition to *Idle* state triggered by *error* signal. These redundant transitions can be eliminated by the application of a refactoring known as *Group States* [17]. During the refactoring, first a new state *Working* is created. Next, three transitions triggered by *error* from *Calculating*, *Encoding*, and *Adjusting* are deleted, and the one from *Decoding* is reconnected to the new state, as well as a transition triggered by *command* from *Idle*. Subsequently, a new state machine is created in *Working*, what makes this state composite. Finally,

Decoding, *Calculating*, *Encoding*, and *Adjusting* are moved together with their transitions to the new state. The transformation is completed by addition of a start symbol. Its effects can be seen in Figure 6.

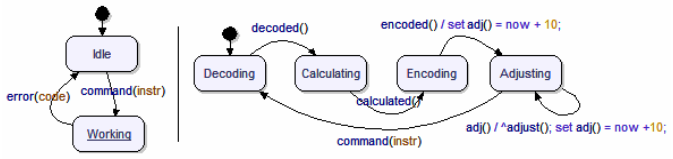


Fig. 6. Two state chart diagrams showing implementation of a default state machine of *InstrumentsController* (left) and *Working* state (right), after applying *Group States*

5.3 Area OIAC – *Replace Method with Method Object*. *Navigator* component of *Satellite* is a compound object, which has as one of its parts an instance of an active class *CollisionDetector*. The class has among others an operation *avoid*, that takes as a parameter an instance of *Collision* class obtained from invocation of *detect* operation. The responsibility of *avoid* is to (1) determine how to avoid the collision and (2) return the result of computation in the form of an instance of *AvoidancePlan* class. The problem with *avoid* is that its operation body is too long, what is an unequivocal symptom of *Long Method* bad smell [5]. However, the operation uses its local variables *dimX*, *dimY*, and *dimZ* in such a way that even after application of *Replace Temp with Query*, its decomposition with the use of *Extract Method* is impossible. Therefore, instead of *Extract Method*, *Replace Method with Method Object* is triggered on *avoid*. The described part of the model before the transformation is shown in Figure 7.

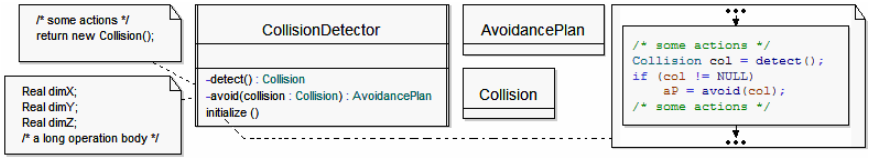


Fig. 7. A class diagram showing a situation qualifying for application of *Replace Method with Method Object*

In the first step, a new passive class is created and named by the operation. Next, an attribute for each temporary variable (*dimX*, *dimY*, and *dimZ*) and the parameter (*collision*) of *avoid* is created in the new class. Then, *Avoid* is given a constructor that initializes *collision* attribute. Subsequently, in the new class a new operation *compute* is created with the body copied from *avoid*. Next, all temporary variables are removed from the body of *compute*, and the body of *avoid* is replaced with one that creates an instance of *Avoid* and calls *compute*. The effect of the transformation is shown in Figure 8. Because all the previous local variables of *avoid* are now attributes, one can easily decompose the operation with the use of *Extract Method*.

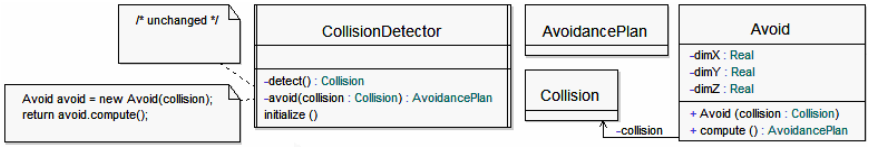


Fig. 8. A class diagram showing *CollisionDetector* after application of *Replace Method with Method Object* on the body of *avoid* operation

It is worth observing that the model after *Replace Method with Method Object* will not work properly if *avoid* invokes any operation of *CollisionDetector* or any operation of any other class accessible from it. In such a situation, Fowler [5] advises to give the new class an attribute for the object that hosts the original operation (the *source object*), initialize it in the constructor, and use it for any invocations of operations on the original class. However, in this case it cannot be done, because *CollisionDetector* is an active class, and passive classes are not allowed to invoke any operations of active ones. Moreover, *avoid* can include neither *output actions* responsible for sending signals nor actions concerning timers, i.e. *timer set* or *timer reset actions*, because these are also not permitted in operation bodies of passive classes.

5.4 Areas ESAC & ESPC – *Hide Delegate*. The refactoring discussed here relates to the Fowler's statement saying that "one of the keys, if not the key, to objects is encapsulation [5]." In general, the less each class in a model needs to know about other classes, the less possible is that a change in one place causes the necessity to adjust other parts of the model, what makes the model maintenance easier and cheaper. For instance (see Fig. 9), let us consider a situation in which a client class (*PlanSupplier*) invokes an operation (*getDestination*) defined on one of the attributes (*plans* accessible via *getDestination*) of a server class (*PlanStorage*).

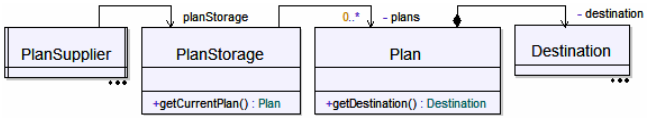


Fig. 9. A class diagram showing a situation qualifying for application of *Hide Delegate*

As the client has to know about the delegate class (*Plan*), each change of the delegate may propagate to the client. This redundant dependency can be removed by placing a simple delegating operation on the server, which hides the delegate. A refactoring that performs this task is known as *Hide Delegate*. First, *getCurrentPlan* is renamed with the use of *Rename Operation* to *getCurrentDestination*. Then, assuming that the most recent plan is always the first one in the *plan* collection, the body of the operation is changed from "*return plan[0]*" to "*return plan[0].getDestination()*". Finally, each statement in the form of "*Destination d = planStorage.getCurrentDestination().getDestination()*" is replaced by "*Destination d = planStorage.getCurrentDestination()*". These statements may occur in bodies of all operations of both passive and active clients of *PlanStorage*, as well as on transitions

in state machines of active ones. After the refactoring, changes become limited to the server and do not propagate to the client (see Fig. 10).

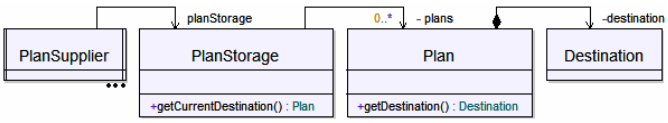


Fig. 10. A class diagram showing the effect of application of *Hide Delegate*

6 Discussion and Further Work

The paper presents an attempt to systematically approach the problem of refactoring of executable UML models. It introduces a notion of refactoring trigger and elaborates categorization of refactorings in the form of refactoring areas based on this notion. Exemplary transformations from each area are presented, and the overall ideas are illustrated on a study executable UML model built in TAU.

The problem of refactoring executable UML models is addressed by Kazato *et al.* in [7], which seems to be the only paper on the topic published so far. The authors deal with refactoring of design models comprising of both structural as well as behavioural parts. The former is expressed by classes and related structural concepts, and the latter is specified with the use of operation bodies implemented in a surface language that maps to action semantics of UML 1.5. Kazato *et al*. defined a set of twenty-eight *basic transformations* of design models, of which various refactorings can be composed. The approach to implementation of refactorings bases on an observation, that each UML model may be represented as an object model comprised of instances of metaclasses of the UML metamodel. Each such an object model is essentially a typed and attributed directed graph, and therefore each basic model transformation can be treated as a graph transformation with rules described as a part of the graph grammar.

Based on the initial study presented in the paper, development of a systematic approach to specification of both executable UML model refactorings as well as associated bad smells in models has been started. In comparison with the work performed by Kazato *et al.*, it is focused on TAU executable models, which are more complex than the stateless ones considered in [7]. Moreover, in place of implementation in a graph-rewriting tool, the refactoring transformations and detection of related bad smells are planned to be programmed in TAU with the use of its metamodel-based COM API.

References

1. Bennett, K.H. and Rajlich, V.T. (2000) 'Software maintenance and evolution: a roadmap', in *Proceedings of the 22nd International Conference on Software Engineering*, 75-87.
2. Boger, M., Sturm, T. and Fragemann, P. (2003) 'Refactoring Browser for UML', *Lecture Notes in Computer Science*, **2591**, 366-377.

3. Chapin, N., Hale, J.E., Md. Khan, K., Ramil, J.F. and Tan, W.-G. (2001) 'Types of software evolution and software maintenance', *Journal of Software Maintenance and Evolution: Research and Practice*, **13**, 3-30.
4. Demeyer, S., Ducasse, S. and Tichelaar, S. (1999) 'Why Unified is not Universal: UML Shortcomings for Coping with Round-trip Engineering', in *Proceedings of 2nd International Conference UML'99 – Unified Modeling Language – Beyond the Standard* (Lecture Notes in Computer Science **1723**), 630-44.
5. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
6. France, R. and Bieman, J.M. (2001) 'Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software', in *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, Florence, 6 – 10 November 2001, 386-395.
7. Kazato, H., Takaishi, M., Kobayashi, T. and Saeki. M. (2004) 'Formalizing Refactoring by Using Graph Transformation', *IEICE Transactions on Information and Systems*, **E87-D**(4), 89-92.
8. Mens, T. and Tourwé, T. (2004) 'A Survey of Software Refactoring', *IEEE Transactions on Software Engineering*, **30**(2), 126-139.
9. OMG (2002) *Unified Modeling Language Specification Version 1.4.2*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?formal/04-07-02.pdf> (29 November 2004).
10. OMG (2004) *UML 2.0 Superstructure Revised Final Adopted specification (convenience document)*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.pdf> (8 April 2004).
11. Parnas, D.L. (1994) 'Software Aging', in *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, 279-287.
12. Pender, T. (2003) *UML Bible*, Wiley.
13. Pigoski, T.M. (1997) *Practical Software Maintenance – Best Practices for Managing Your Software Investment*, John Wiley & Sons.
14. Pollet, D., Vojtisek, D. and Jézéquel, J.-M. (2002) 'OCL as a Core UML Transformation Language', WITUML Position Paper at *16th European Conference on Object-Oriented Programming*, Málaga, 10-14 June 2002.
15. Porres, I. (2003) 'Model Refactorings as Rule-Based Update Transformations', Technical Report Series No. 525, Turku Center for Computer Science, Finland.
16. Roberts, D.B. (1999) 'Practical Analysis for Refactoring', PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.
17. Sunyé, G., Pollet, D., Le Traon, Y. and Jézéquel, J.-M. (2001) 'Refactoring UML Models', *Lecture Notes in Computer Science*, **2185**, 134-148.
18. Van Gorp, P., Stenten, P., Mens, T. and Demeyer, S. (2003) 'Enabling and using the UML for model driven refactoring', in *Proceedings of the 4th International Workshop on Object-Oriented Reengineering (WOOR)*, Darmstadt, 21 July 2003.
19. Van Gorp, P., Stenten, P., Mens, T. and Demeyer, S. (2003) 'Towards automating source-consistent UML refactorings', in *Proceedings of the 6th International Conference on UML – The Unified Modeling Language*, San Francisco, 20 – 24 October 2003.
20. Van Gorp, P., Van Eetvelde, N. and Janssens, D. (2003) 'Generating Refactoring Implementations from Platform Independent Metamodel Transformations', in *Proceedings International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI 2003)*, Luxembourg, 27 – 28 November 2003.
21. Van Gurp, J. and Bosch, J. (2002) 'Design Erosion: Problems & Causes', *Journal of Systems & Software*, **61**(2), 105-119.
22. Zhang, J., Lin, Y. and Gray, J. (2004) 'Generic and Domain-Specific Model Refactoring using a Model Transformation Engine', *Model-driven Software Development – Research and Practice in Software Engineering*, accepted for publication in 2005.

Run-Time Monitoring of Behavioral Profiles with Aspects

Kimmo Kiviluoma[2], Johannes Koskinen[1], and Tommi Mikkonen[1]

[1] Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{johannes.koskinen, tommi.mikkonen}@tut.fi
[2] Solita Oy
Satakunnankatu 18 A, 33210, Tampere, Finland
kimmo.kiviluoma@solita.fi

Abstract. Dynamic program behavior is often an essential part of a system architecture. Verifying the behavior is a crucial yet often an intractable part of the testing. Hence, it is of great concern to find means to facilitate the testing of dynamic behavior. This paper studies one approach to behavioral monitoring. We have used the concept of behavioral profiles to specify the desired program behavior with UML. Provided with a behavioral profile created with a CASE tool, we are able to automatically generate AspectJ aspects that weave a monitoring concern into Java program code.

1 Introduction

While architecture necessarily addresses the structure of a system, also behaviors of the system can be architecturally significant. For instance, it is essential that clients do not communicate with each other directly when the client-server architecture is used. Such rules can be embedded in architectural descriptions, for instance, which in principle enforces them.

In practice, documenting rules in a separate document is not enough in the general case. Instead, the rules should be such that tool support can be implemented for specifying them at the level of architectural descriptions and that a mechanism can be created for monitoring them in the implementation.

In this paper, we introduce an approach where UML metamodel extensions are used for the first purpose, i.e., defining the architecturally significant behaviors. With the extensions, we define *behavioral profiles*, which can be used for defining architecturally significant sequences of behavior. For the second purpose, we use aspect-oriented techniques. The goal is to generate an aspect (or aspects) that are based on behavioral profiles. Then, the aspects are woven to a system that has been designed to follow the profiles. Executing the system then reveals the violations of architecturally significant behaviors at run-time.

The rest of this paper is structured as follows. Section 2 introduces UML profiles, including the behavioral variant we use for representing architecturally

significant behaviors. Section 3 discusses run-time monitoring with behavioral profiles, and Section 4 gives an example on the use of the approach. Section 5 discusses the most important findings on implementing the approach, and Section 6 addresses related work. Finally, Section 7 concludes the paper with some final remarks.

2 UML Profiles

2.1 Profile Mechanism

The profile mechanism of UML has been created to adapt UML for different purposes and domains by introducing *stereotypes*. The mechanism gives a terminology, notation and semantics used in the modeled domain. In addition, profiles can be used to add domain specific constraints and mapping rules between two models. The constraints are applied to the instances of stereotypes. Thus, a profile essentially defines a domain specific modeling language.

New elements can be added using the profile mechanism to extend UML. In the profile, classes having a stereotype «stereotype» create new stereotypes to be used in a model. The stereotypes alter or extend semantics of the existing UML metaclasses. In addition, they can define new metaclasses with new metaattributes.

For example, UML can be tailored to be used with a resource intensive domain by defining a new profile for the domain. The profile defines an additional metaclass *Resource* that extends a regular class and adds a new metaattribute to specify the type of the resource (*type*). The new resource element can now be used in place of a regular class to model the different resources of the domain. The profile and its usage are illustrated in Fig. 1

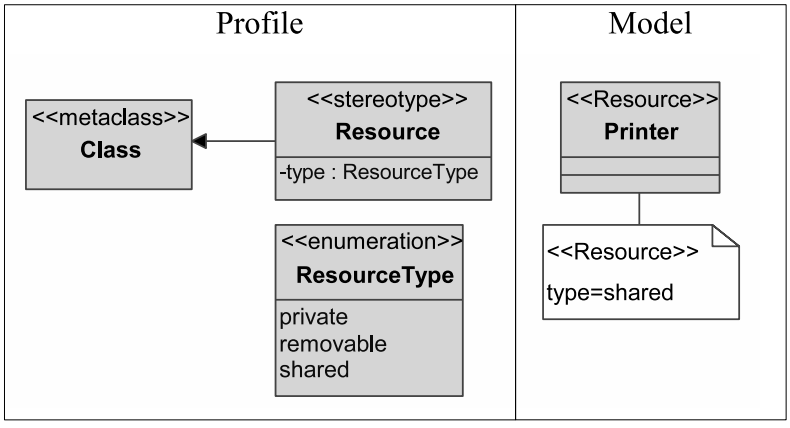


Fig. 1. Defining and using a simple resource profile

2.2 Behavioral Profiles

In addition to structural relationships, there usually is important interaction between the components. For example, using a framework may require following certain behavioral rules like establishing a session to a server before using the services provided by the server in a client-server system. Also, *design patterns* [1] usually have specified interaction patterns.

Architecturally significant behavioral rules can be presented as a UML profile. Our approach to express the rules is referred to as *behavioral profile*. The concept of the behavioral profile is built on top of UML metamodel by adding three new stereotypes:

- *«ClassRole»* is used to define the roles of the participants (like *client* and *server*) in the profile. A role can also be inherited from another role. For example, *UserServerRole* can be an extension of *ServerRole*.
- *«OperationRole»* defines that the operation inside a class role is used as a role.
- *«AttributeRole»* defines an attribute role that is used for e.g. constraints or invariants.

A behavioral profile consists of class diagrams containing role definitions and behavioral rules in the form of sequence diagrams. The latter are more focused on significant parts of behavior than specifying an exact execution path for the whole program. As UML 2.0 is chosen for the notation of behavioral profiles, creating the profiles can be carried out with an UML CASE tool, and behavioral models (e.g. results from reverse engineering process) can be automatically checked against behavioral profiles with proper tool support.

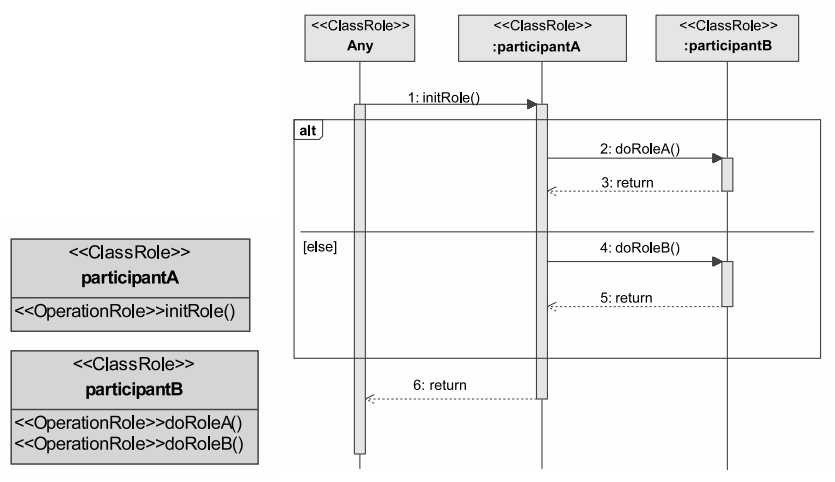


Fig. 2. The notation of behavioral profile

A sample behavioral profile is shown in Fig. 2. The profile states that after a participant (*Any* refers to any object in the system) has made a method call marked with role *initRole* to a class in role *participantA*, either *doRoleA* or *doRoleB* in *participantB* has to be called before the end of the method. Any additional calls can occur in method *initRole*.

The profile roles are expressed with stereotypes in the model. So, the mapping between roles and actual classes is done by using a corresponding stereotype for the class. A sample profile with a corresponding model is shown is Fig. 3.

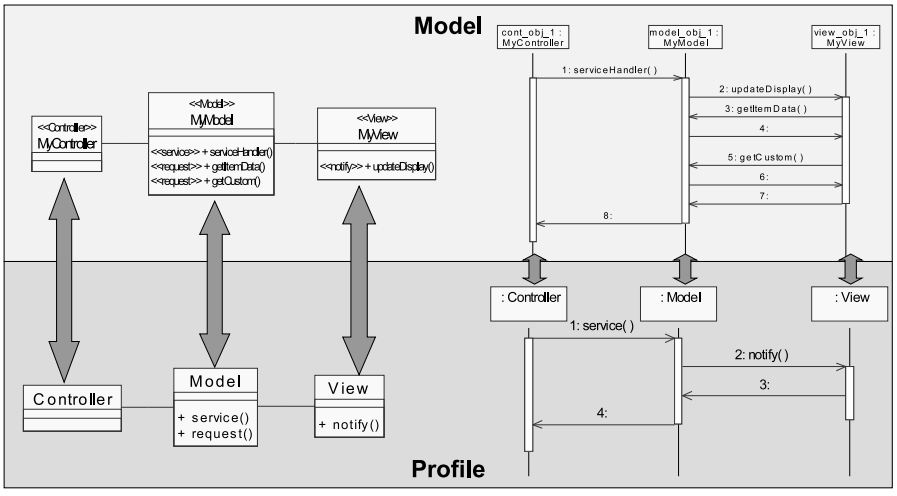


Fig. 3. A behavioral profile with a corresponding model

3 Run-Time Monitoring with Behavioral Profiles

3.1 Overview

Often API or framework providers give some instructions on how to use the API in a separate document or in Java API documentation as the interface (method signatures) itself does not give enough details on allowed call sequences, for instance. This extra documentation may be complex, and often it is not thoroughly read. For this reason, we introduce a way for API providers to make sure that their API is correctly used. This is done by adding a monitoring aspect to the API client program. Similarly, architecturally important behavioral rules can be enforced in a system or the correct use of a resource (allocating, freeing) can be insured.

To monitor the behavior of a program at run-time, we use behavioral profiles to specify rules for program execution. The rules define what occurrences are allowed to happen at a certain point of execution.

In general, as the behavioral profiles can be specified with UML, the rules for program execution are limited only by the expressive power of UML itself. However, we have selected a small commonly used subset of UML 2.0 that can be used to specify the rules for the baseline tool implementation.

We have chosen Java as the language for the monitored program. Technically, the monitoring is performed by augmenting the Java source code with aspects that are automatically generated from the behavioral profile. The aspects are implemented with AspectJ language.

3.2 Selected Tools

To build a proof-of-concept environment for the run-time monitoring, we have used existing UML tools. The tool set consists of a CASE tool, a UML model processing platform and an aspect concept implementation for Java language.

xUMLi [2] is a CASE-tool independent processing platform following the UML metamodel. In addition to metamodel interface, the OCL interpreter is provided. Processing components for the platform can be written using e.g. Python or Java. UML models and diagrams can be imported from or exported to CASE-tools (like Rational Rose).

Rational Rose [3] was chosen as the tool for drawing behavioral profiles due to its wide-spread use in the industry and easy access to UML metamodel with xUMLi. The biggest downside of its use is the absence of support for UML 2.0 in the latest version.

AspectJ language implements the aspect-oriented paradigm [4]. It aims at managing crosscutting concerns, i.e., concerns that span across multiple modules. AspectJ augments Java language with a few new structures to help to manage crosscutting concerns. Crosscutting concerns are separated into new modularization units, which are called aspects. In addition to aspects, AspectJ introduces new structures like pointcuts and advices. In short, the idea is to use pointcuts to select a set of identifiable points in the execution of the program (join points) and use advices to attach code to those points. [5]

3.3 Development Process

In order to use UML profiles to specify behavioral rules, a translation between UML and AspectJ domains is required. The translation process is illustrated in Fig. 4. The translation can be regarded as an operation that has two inputs and one output. The first input contains bindings between roles used in the profiles and actual program classes. For the bindings, a class diagram with played roles marked with stereotypes is required. The other input contains behavioral profiles used for the execution rules (*Profile descriptions*). As a result of the operation, AspectJ code is generated.

The process starts with analyzing the profiles. Based on the analysis, necessary infrastructure is created inside the generated aspects. Then the generated aspects are specialized to be used with the actual classes. For the specialization,

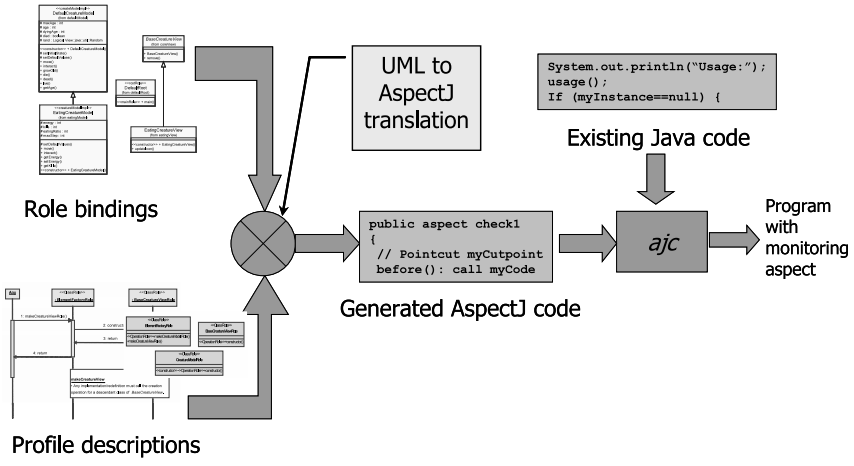


Fig. 4. Process flow

role binding information from the class diagrams is used to generate correct pointcuts for the actual classes.

After the translation process, the generated aspects augment the program code as they are compiled with the existing program code. As a result, we get a program that contains a monitoring concern. As the augmenting is done compile time, it is not possible to apply rules with behavioral profiles at run-time.

As the augmented program is executed and a rule violation occurs, a *RuntimeException* can be thrown, the program execution can be terminated or the details of the violation can be written in a configured log file.

3.4 Limitations

In our current solution, occurrences in the profile are only allowed to be synchronous messages. These correspond to Java method calls.

For now, we do not allow UML *CreationEvents* in the profiles. They could be easily supported with some additional work. Also, *DestructionEvents* are not supported due to characteristics of the Java language.

One particular detail of this work is that we are checking the calls between classes, not class instances. The generated aspects contain pointcuts for method calls of certain classes, not for specific objects created runtime. All the rules apply to all objects of a certain class. Therefore, operation sequences that have significance for some particular instance only fall beyond the scope of this work. A further detail is that binding roles to objects that only exist in a limited scope during program execution and making the generated aspect to check the rules only for those objects is not addressed.

4 Example

4.1 Sample Pattern

As an example, we have chosen a well-known GoF pattern, *Observer*, and to be more specific, the protocol how the observers receive the subject's data as the subject undergoes a change in state in the pull model *Observer* case.

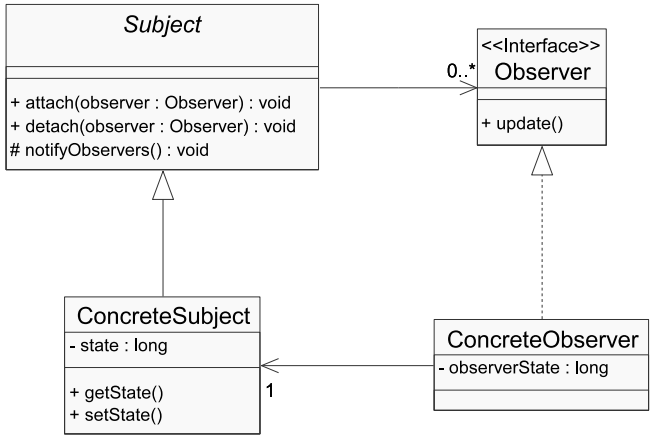


Fig. 5. Java implementation of the observer pattern

Prerequisite for the behavioral monitoring is that there are one or more observers registered to the subject. As the *ConcreteSubject's* setter method is called changing its state, the *ConcreteSubject* calls *notifyObservers* method inherited from the abstract *Subject* class. The method then loops over all registered observers calling the *update* methods in the *Observer* interface. *ConcreteObserver* objects, which implement the *Observer* interface, will then query the *ConcreteSubject* and receive the new state.

From the *Observer* pattern we can recognize two class roles, subject and observer. Classes that contain the data of interest belong to the subject role, and classes that are dependent on the subject's data and need to be notified when it changes belong to the observer role. Fig. 6 shows the two class roles as *SubjectRole* and *ObserverRole*. Within the class roles, there are several operation roles, *getter*, *setter* and *notify* in *SubjectRole* and *update* in *ObserverRole*. Any method in the class that belongs to the subject class role may have any of its methods in the above mentioned three method roles. A method may belong to many roles at the same time. Similarly, any method in *ObserverRole* may be in *update* operation role.

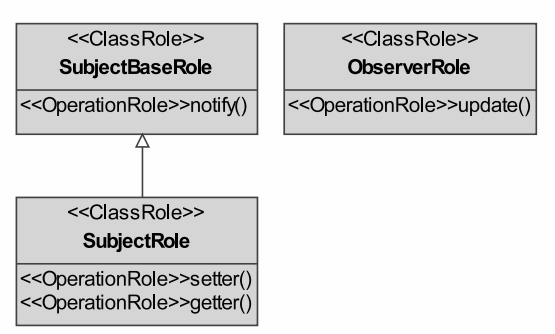


Fig. 6. Roles used in the observer profile

4.2 Behavioral Profile

The behavioral profile itself is presented in Fig. 7. It specifies the communication between *SubjectRole* and *ObserverRole* using operation roles. Of course, there may be multiple classes in the observer role. Because the behavioral profile is drawn between class roles, it does not express the order in which the observers are notified. In case of *Observer* pattern, the order does not concern us. However, the method in *notify* role is responsible for updating all the observers by calling their method in *update* role.

Instead of drawing the behavioral profile, it is, of course, possible to reverse engineer a sequence diagram from existing code and modify it so that it brings out the essential classes and their interaction.

The profile expresses only the order of the calls present in the profile. For example, the method in the *update* role could call other subject's methods before or after the call to the method in the *getter* role. This kind of behavior could be prohibited with *critical* combined fragment.

The monitoring is activated with the *setter* call and deactivated as the *setter* call returns. The *notify*, *update* and *getter* calls that do not happen between the *setter* call and its return are ignored. The first call in the profile always activates the monitoring.

4.3 Mapping Roles to Model

After we have defined the rules for monitoring by drawing a behavioral profile for the interaction of the roles, we still need to somehow map the class roles and operation roles to the actual classes and methods. This is done by reverse-engineering the monitored program to get a simple class diagram and putting the role names as stereotypes for the classes and methods. Fig. 8 shows how the mapping is done in the case of our example. Notice that the *ObserverRole* is mapped to the *Observer* interface and not the *ConcreteObserver* class. This is due to the fact that in AspectJ the method call pointcut picks a method call based on the static type used to access the method.

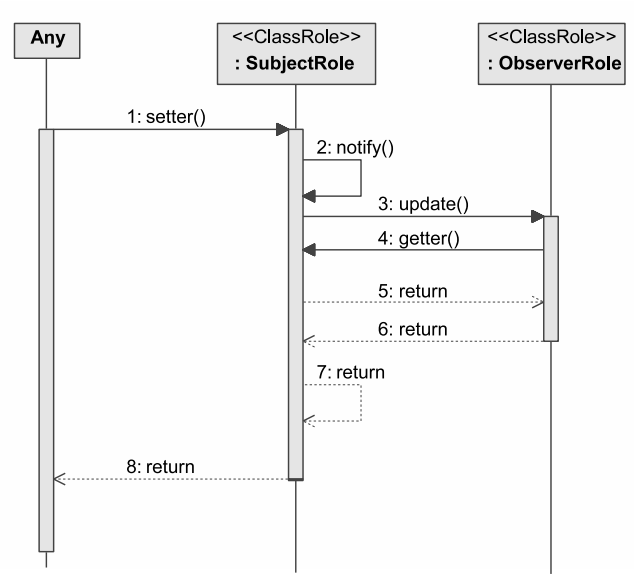


Fig. 7. The observer profile

The subject class role is divided into two actual roles, the *SubjectRoleBase* and its extension *SubjectRole*. The actual role in the *Observer* pattern is *SubjectRole*. The *notify* role is mapped to the *notifyObservers* method in the abstract *Subject* class as it contains the implementation of the method. The *SubjectRole* extends *SubjectRoleBase* inheriting its mappings. It is required that all the operation roles are mapped to some methods in the class that is in *SubjectRole*.

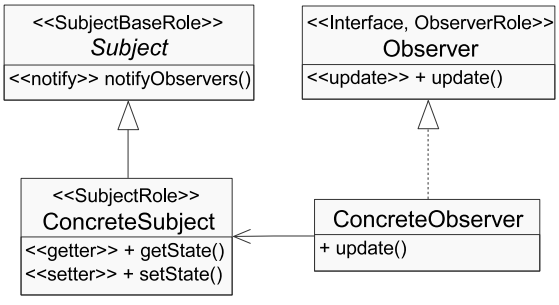


Fig. 8. Role bindings for the observer pattern

4.4 Aspect Generation

To generate the aspect used for monitoring, the location of the diagrams in Fig. 7 and Fig. 8 is given to the aspect generator. The program imports the diagrams and according the class diagram information it generates mappings between roles and the actual Java classes and methods. The mappings and the sequence diagram parts of the imported profile are used to generate an AspectJ code to be woven inside the Java program. To illustrate mappings between the roles and the actual Java classes, a sample of generated code is shown in Fig. 9.

```
// SubjectRole.notify
before(): call(* ConcreteSubject.notifyObservers(..))
      && withincode(* ConcreteSubject.setState(..)) {

      checkAndChangeState(1,2);
}
```

Fig. 9. An example of the generated aspect

4.5 Sample Execution

The implementation of the *Observer* pattern was tested using *JUNIT* utility. The code in the *setState* method (shown in Fig. 10) was changed to call *notifyObservers* from zero to two times. As the *Observer* profile states (refer Fig. 7), only single *notify* call should be accepted. When the number of the *notify* calls differed from one, the run-time exception (see Fig. 11) was thrown referring the violating line in *setState*.

```
public void setState(long state) {
      this.state = state;

      // run this line 0-2 times
      notifyObservers();
}
```

Fig. 10. Code fragment for the sample execution

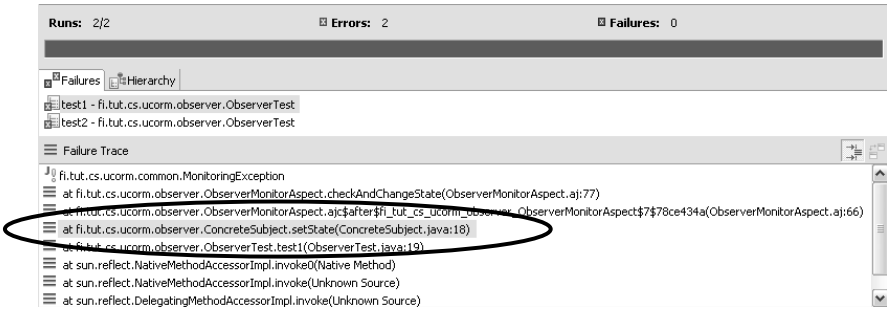


Fig. 11. An exception shows the violating line

5 Findings

While examining several approaches to implement desired monitoring functionality, a number of observations were made. One of the first approaches was to treat the whole profile as a set of ordered pairs of calls and to check before the latter call that the first call has been done. That approach was quickly bypassed because the method can be called several times in a profile. This clearly indicates that it is necessary to create a state machine inside an aspect and make the specified method calls change the internal state of the aspect. This has been taken into account in the reference implementation.

Before each method, the state must be checked, and in case of an illegal state transition, an exception is thrown. We allow the methods in the diagram to be called if they are not part of the sequence, i.e., all the calls to methods present in the profile are allowed if the first call in the profile has not been made. The first call in the profile activates the monitoring.

To make sure that the calls are made from a certain method to another we had to make pointcuts that check two things: a specified method is called and a specified method of a specified class makes the call. No indirect calls are allowed. In the case of *Found Messages*, the caller is not checked, and with *Lost Messages* the callee is not speficied in the pointcut.

The returns from methods can be checked as well. By including them in the profile, we can ensure that no undesired calls are done before the return. Imagine a case in which we want method *method1* of class *A* to call method *method2* of class *B* only once. This situation is described in the profile on the left side of Fig. 12. The return from *method1* insures that *method2* is called only once. If the return operations would be ignored and not drawn in the profile, the situation in the sequence diagram on the right side of Fig. 12 would be possible. In this case the monitoring would be stopped after the first *method2* call and everything is allowed afterwards including second call to *method2*.

An obvious consequence of the limitations discussed above is that in our example, the given profile does not work in a situation where one subject has multiple observers. If multiple observers for one subject were to be supported,

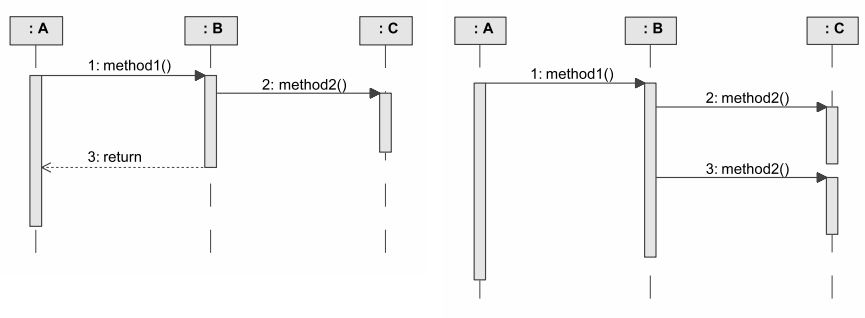


Fig. 12. A return drawn in the profile on the left insures that the situation on the right can not happen

this would also have to be expressed somehow in the profile or in the mappings (or both).

It was further noticed that combined fragments require careful treatment as using them within each other may change the semantics of the situation dramatically. For example, an alternate combined fragment behaves differently if it is enclosed by a critical or strict combined fragment. In perspective of implementation, this means that the state machine generating the aspect will become rather complex. The generating code may need an internal state machine to cope with the combined fragments. Depending on the state of the state machine, the operation decides which occurrences are allowed to happen at certain points of execution.

Some minor drawbacks of using AspectJ did come out. AspectJ's capabilities are not completely in line with the properties of UML 2.0 sequence diagrams. For instance, AspectJ inherits the Java problems with object destruction. A finalizer execution is possible to intervene, for instance, with an execution pointcut, but the time between the moment the object becomes unreachable and execution of finalizer is arbitrary. In addition, the finalizers may not be executed at all in some cases [6]. Therefore, the sequence of object destructions in the behavioral profile is impossible to verify with AspectJ and Java.

More importantly, with AspectJ it is difficult to verify if a method call is ever done. If a profile requires that class in role *A* calls two methods, *method1* and *method2*, of class in role *B*, after first method call to *method1* it is not easy to raise an error if the second call *method2* never seems to come. This situation is shown in Fig. 13.

This problem lies in the fact that the profile does not define how much time can be spent between the two method calls. This could be done with UML 2.0 *Duration* and *Time Constraints*. Using them with AspectJ would be a bit inaccurate as AspectJ does, even though just a little, slow down the program execution. Yet, with making a bit higher upper limits to time spend between the calls, this problem could be sufficiently solved. Technically, one possibility would be to create a time monitoring thread inside the aspect that would check

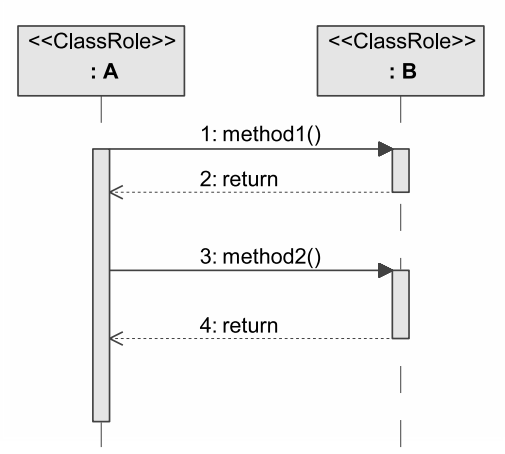


Fig. 13. A situation where checking the second method call is difficult

the state of the aspect after a certain time was exceeded. However, this was not implemented.

A simple solution to this problem is to add a surrouding method call and a return to profiles. An example of this is *setState()* in Fig. 7. As the execution returns from the method, it is possible to check if all the preceding calls are made. This is certainly not an infallible approach as the return from the method might never happen, e.g., in a case of an eternal loop between the methods.

With virtual machine shutdown hooks (java.lang.Runtime.addShutdownHook) [7] it would be possible to report if a profile monitoring was ongoing and its current state at the time of program termination. This could be used to notice that a certain call was never made. Unfortunately, not even shutdown hooks are always executed at the time of program termination.

AspectJ decently met our needs but it may not be the best possible tool implementing runtime monitoring of behavioral profiles. Instead, other approaches such as using Java Debug Interface (JDI) may turn out to be a better alternative.

By slightly altering the development process, the presented system might also be used for automatic recovery from an error situation in a system or maybe even writing aspects with UML.

6 Related Work

To support structural software architecture design, Selonen and Xu [8] have introduced a concept of *architectural profiles* that specify architecturally important or interesting structural relationships between components. A design validity model can be checked against the rules specified in a profile. The checking is done using a special tool (*ArtDeco*) in design phase, so no running code is required. The concept has been used to support software maintenance [9].

In [10] Richters and Gogolla present an aspect-oriented approach for monitoring UML and OCL constraints. Constraints are given in a UML design model using OCL. The constraints include class invariants and pre- and postconditions. A special tool, *USE*, is used to validate the UML model and to generate a monitor aspect. The aspect generates information on the behavior of the application which is later on used for behavior validation in the USE tool. In addition, the tool generates a sequence diagram to show a trace of operation calls in a monitored system. While constraints are given within UML class diagram, they are still in textual form. In addition, the constraints refer to actual Java classes, so there is no role based validation mechanism. This restriction makes distributing behavioral constraint libraries along with programming interface or framework more difficult.

In [11] Groher and Schulze present an approach to support modeling aspects in UML. The aspects can be modeled using an UML CASE tool and corresponding AspectJ code skeletons are generated. The skeletons are used to offer an automated mapping from design models to programming models.

Yan et al. [12] present a technique called *DiscoTect* to construct an architectural view from an executing system. The technique uses a monitoring tool (*Trace Engine*) to get filtered trace information. The information is fed to runtime event pattern recognition engine (*State Engine*). The engine outputs a set of architectural operations to build the view using *Architecture builder*. The technique also includes a language to define mappings that describe how the events are interpreted as architectural operations. Even though the technique is used to construct architectural views, it may be possible to change the mappings to support behavioral constraints. However, the notation for State Engine is textual, so a conversion between UML and the mapping language is required.

7 Conclusions

While the description of software architecture necessarily addresses structure of the system, also behavioral properties bear significance. In this paper, we have introduced a way to denote architecturally significant behaviors in terms of behavioral profiles using UML as the notation. Furthermore, we then introduced a way to generate code based on behaviors, resulting in an option to introduce monitoring regarding the architecturally significant behavior. While in this paper we used AspectJ as the implementation technique for monitoring, also other options, such as Java Debug Interface (JDI), could be used. We also gave an example, where the different concepts were addressed with a simple pattern.

Obviously, a lot of future work remains to be done. Most importanty, the monitoring system should be enhanced to cope with class instances, not just classes. Further, it would be interesting to study if it is feasible to automatically introduce behavioral profiles for some library facilities that require monitoring. This would then ease debugging in the case of errors. Moreover, studying if behavioral profiles could be used as a generic way to represent aspects at the

level of architecture could result in a way to connect aspects to UML in a novel fashion.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
2. Airaksinen, J., Koskimies, K., Koskinen, J., Peltonen, J., Selonen, P., Siikarla, M., Systä, T.: xUMLi: Towards a tool-independent UML processing platform. In: Proceedings of the Nordic Workshop on Software Developement Tools and Techniques, Copenhagen, Denmark, IT University of Copenhagen (2002) 1–16
3. Rational Software http://www-306.ibm.com/software/rational: Rational Rose. (2005)
4. Kiczales, G., Lamping, J., Mendhekar, A., Maede, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97), Springer-Verlag (1997) 220–242
5. AspectJ Team http://eclipse.org/aspectj/: (The AspectJ Guide)
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc. (2000)
7. Sun Microsystems, Inc. http://java.sun.com/j2se/1.5.0/docs/api/: (Java 2 Platform Standard Edition 5.0, API Specification)
8. Selonen, P., Xu, J.: Validating uml models against architectural profiles. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2003) 58–67
9. Riva, C., Selonen, P., Systä, T., Xu, J.: Uml-based reverse engineering and model analysis approaches for software architecture maintenance. In: ICSM, IEEE Computer Society (2004) 50–59
10. Richters, M., Gogolla, M.: Aspect-oriented monitoring of UML and OCL constraints. In Akkawi, F., Aldawud, O., Booch, G., Clarke, S., Gray, J., Harrison, B., Kandé, M., Stein, D., Tarr, P., Zakaria, A., eds.: The 4th AOSD Modeling With UML Workshop. (2003)
11. Groher, I., Schulze, S.: Generating aspect code from UML models. In Akkawi, F., Aldawud, O., Booch, G., Clarke, S., Gray, J., Harrison, B., Kandé, M., Stein, D., Tarr, P., Zakaria, A., eds.: The 4th AOSD Modeling With UML Workshop. (2003)
12. Yan, H., Garlan, D., Schmerl, B.R., Aldrich, J., Kazman, R.: Discotect: A system for discovering architectures from running systems. In: ICSE, IEEE Computer Society (2004) 470–479

UML 2.0 Can't Represent Architectural Connectors

Jorge Enrique Pérez-Martínez[1], Almudena Sierra-Alonso[2]

[1]Departamento de Informática Aplicada, Universidad Politécnica de Madrid,
Crta. de Valencia, Km.7, 28031 Madrid (Spain)
jeperez@eui.upm.es
[2]Escuela Politécnica Superior, Universidad Autónoma de Madrid,
Crta. de Colmenar, Km. 15, 28049 Madrid (Spain)
Almudena.sierra@uam.es

Abstract. Recently, OMG has published a set of documents that will constitute the future UML 2.0 specification. One of the goals of this new version of the language is that UML can represent the software architecture of an application. In this work we show some shortcomings of the language when it is used to represent the connector concept in a software architecture. Those shortcomings are basically caused because in UML 2.0 the connectors are not first class entities, as components are. In this work we will show that the new metaclass *Connector*, defined in the packages *InternalStructures* and *Components,* can not support the semantics that a connector has in some architectural styles.

1 Introduction

UML 1.x [14] has become the standard for representing the software products obtained in the various activities of a software development process. For this reason, it is not surprising that there have been attempts to use UML 1.x to represent the software architecture of an application. However, the language is not designed to represent syntactically and semantically the elements of software architectures. Some works analyzing this problem are [4, 5, 7, 9, 12, 20, 21, 22].

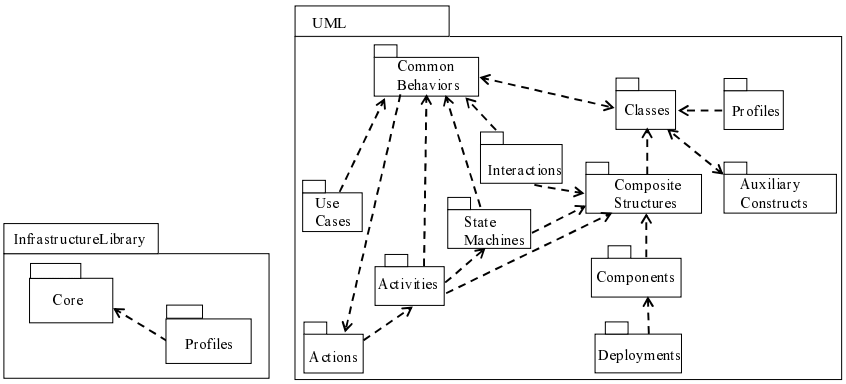
Recently, OMG (Object Management Group) has published a set of documents that will constitute the future UML 2.0 specification [15, 16, 17, 18]. As Douglas says [3]: "Two main forces drive the RFP`s requirements: scalability and architecture.". In fact, in UML 2.0 Superstructure RFP [13] it is indicated: "However, the ability to model architectures is a common requirement for most software domains and, consequently, should be part of the core modeling capabilities of UML rather than being limited to a profile".

If we consider that UML 2.0 should have the fundamental features of an ADL (Architecture Description Language), then it should have abilities to describe the connectors of any architectural style. However, as we will demonstrate in this work, UML 2.0 has not defined any constructor in its metamodel to describe this concept. In this sense, the metaclass *Connector* is limited to connect components and it does not support the connector semantics found in some architectural styles.

The rest of the paper is organized as follows. In Section 2 we describe the software connectors in UML 2.0. In Section 3 we describe the problems we found to characterize a software connector in different architectural styles using UML 2.0. In Section 4 we point out some possible solutions based on tools of UML to extend the language. Section 5 presents some conclusions and future work.

2 Software Connectors in UML 2.0

The specification of UML 2.0 is provided in two large packages: *InfrastructureLibrary* (Figure 1a) and *UML* (Figure 1b).



(a) (b)

Fig. 1. The InfrastructureLibrary and UML packages

The concept of connector is specified by the metaclass *Connector* inside the package *CompositeStructures::InternalStructures*. In this package a connector is defined to specify the link allowing communication among two or more instances. Figure 2 shows some metaclasses contained in the package *CompositeStructures::InternalStructures*.

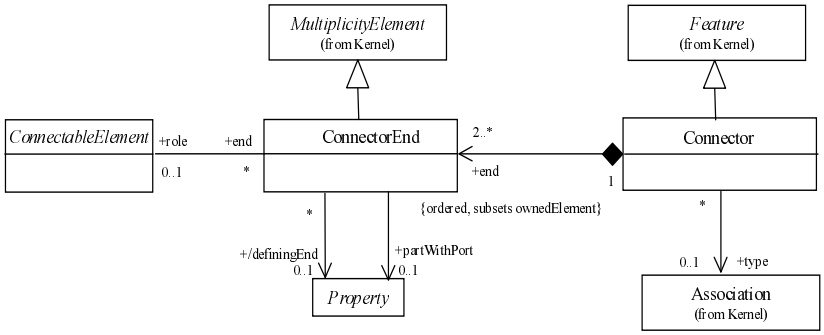


Fig. 2. Definition of the metaclass Connector in the package CompositeStructures::InternalStructures

The package *Components* (Figure 3) extends the concept of connector of the package *CompositeStructures::InternalStructures*, adding the attribute *kind* whose value is an enumerated type with the values *delegation* and *assembly*. A delegation connector links the external contract of a component with the internal parts of the component implementing that behavior. A delegation connector can only be defined between ports of the same type. An assembly connector links two components establishing that one of the components provides the services needed by the other one. Connectors of this type are defined from an interface or port required to an interface or port provided.

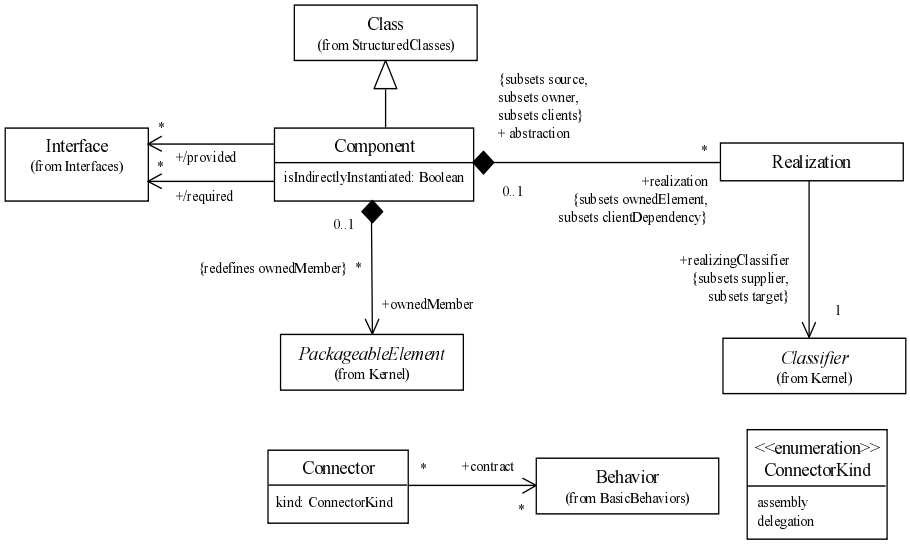


Fig. 3. The Components package

3 Software Connectors in UML 2.0 and Software Connectors in the Architectural Styles

In this section we analyze whether the metaclass *Connector* defined in UML 2.0 supports the semantics of software connectors of some architectural styles described in the literature. As an example we will study this problem for the architectural styles pipe&filter and C2.

3.1 The Pipe&Filter Style

"The pattern of interaction in the pipe-and-filter style is characterized by successive transformations of streams of data. Data arrives at a filter, is transformed, and is passed through pipes to the next filter. A pipe is a connector that conveys streams of data from the output port of one filter to the import port of another filter. Pipes act as unidirectional conduits, providing an order-preserving, buffered communication channel to transmit data generated by the filters" [2].

To describe this style with UML 2.0, filters can be represented as instances of the metaclass *Component,* so that the ports associated with a component (instances of the metaclass *Port*) are port-in or port-out. Pipes can be represented as instances of the metaclass *Connector* whose connector-ends are instances of the metaclass *Port* (which is a subclass of the metaclass *ConnectableElement*). In Figure 4 an architecture is represented with the pipe&filter style.

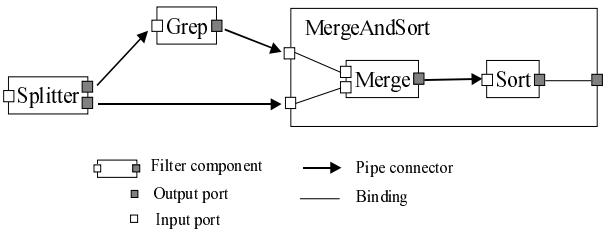


Fig. 4. A system describes with the pipe&filter style

Figure 5 is an attempt to represent that architecture with UML 2.0. Although we have to refine the metaclass *Port* to characterize the type of port (input or output), the basic problem is the representation of a pipe. A pipe has features for buffering and synchronization: if a filter writes into a full pipe the filter is blocked; if a filter reads from an empty pipe the filter is blocked. Furthermore, in some substyles data have an associated type. Lastly, in the pipe&filter style, topological restrictions can be imposed to generate, for example, a pipeline.

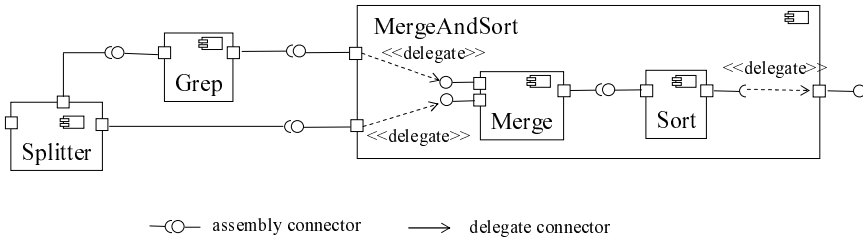


Fig. 5. The pipe&filter system of Figure 4 represented with UML 2.0

The first problem stated in Figure 5 is that the semantics of the connector is not explicit. The connectors which appear in Figure 5 do not indicate the communication protocol between components. In fact, from that figure we can not affirm whether the components communicate by message passing, procedure call or events announcement. The notation used in Figure 5 only indicates that the services required by a component are provided by another one (assembly connector) or that the services provided by a component are really implemented by an internal element of this component (delegate connector). In UML 2.0 the semantics of a connector is limited to indicate the elements that it connects (instances of *ConnectableElement*). In this sense, UML 2.0 is not able to describe the semantics of a pipe as opposed to other ADLs as Darwin [10], Unicon [23] or Wright [1] do.

We could use the <<pipe>> label (as stereotype) on the assembly connectors in figure 5. However, there are two objections to use this strategy: 1) Using UML 2.0 we should be able to represent software architectures without having to turn to profiles; 2) UML 2.0 cannot express the semantics of a pipe because a connector cannot have any behavior associated. A stereotype on the metaclass connector (<<pipe>>) can establish new constraints but not a new semantics.

3.2 The C2 Architectural Style

"The C2 architectural style can be informally summarized as a network of concurrent components hooked together by message routing devices" [11]. The key elements of architecture C2 are components and connectors. Both have a defined top and bottom domain. The top domain of a component specifies the set of notifications to which the component responds and the set of requests sent by the component. The bottom domain specifies the set of notifications sent by the component and the set of requests to which it responds.

A connector can be connected to any number of components and/or connectors. Connectors are responsible for routing messages and potentially multicast them. A secondary responsibility of them is message filtering. Connectors can provide the following policies for filtering and delivery of messages: no filtering, notification filtering, message filtering, prioritized and message sink.

The first problem to consider is that, in C2, a connector can be connected to any number of connectors, and not only components. In UML 2.0 the ends of a connector

must be constructors of the type *ConnectableElement* (see Figure 2). UML 2.0 only defines the following metaclasses of this type: *Property*, *Variable*, *Port* and *Parameter*. Since in UML 2.0 the metaclass *Connector* is not of type *ConnectableElement*, a connector cannot be connected to other connectors. This makes it impossible for *Connector* or any of its stereotypes to represent a C2 connector.

Secondly, a connector in UML 2.0 is only a type of association. In C2, a connector, like a component, can be formed by components and connectors. However, the metaclass *Connector* is not a *PackageableElement* and consequently it cannot be contained by any component nor it can contain other connectors.

Finally, consider the architecture indicated in Figure 6a. In the C2 style, the component A will send requests to the components B and/or C and these will send notifications to A.

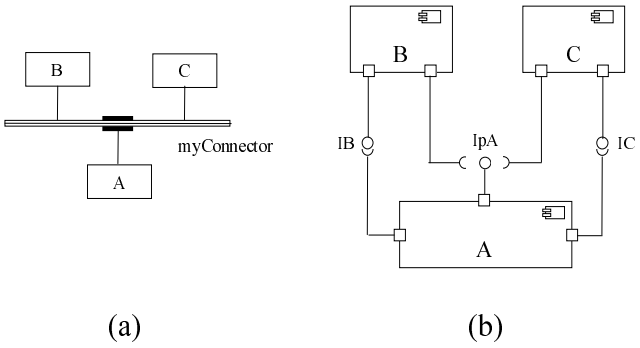


Fig. 6. Example of a C2 architecture (a) and its representation in UML 2.0 (b)

Figure 6b shows how the architecture of Figure 6a could be represented using UML 2.0. The principal problem is the same as in the case of the pipe&filter style: the connectors do not have an explicit semantics. In this case, the C2 connectors realize filtering policies over the messages that they receive. This can not be indicated with UML 2.0.

In the previous example we assume that the connector (*myConnector*) has an associated filtering policy (*no filtering*) that indicates its semantics: sending the request from the component A to the component B and the component C and sending the notifications generated by B and C to the component A. With the representation of Figure 6b, we can not deduce this communication policy. In that figure it is indicated that the component A needs the services provided by components B (interface IB) and C (interface IC). If these two components provide the same service that A requires, for example S, when A invokes the service S: will services from B and C be invocated, as it would be in C2? Will only one of them be invocated? In this case, which one?

4 Stereotypes and New Members of the UML 2.0 Family

Considering the problems stated above, it looks like that the metaclass *Connector* of UML 2.0 as it is defined can not represent a software connector as it is defined by the software architecture community. Therefore it is necessary to extend the language. We can extend UML 2.0 in two ways:

- We can define a new dialect of UML 2.0 by using profiles.
- We can specify a new language related to UML 2.0 by reusing part of the *InfrastructureLibrary::Core* package and augmenting it with appropriate metaclasses and metarelationships.

In regard with the definition of a profile, some cases are already described in the literature. For example, in the work by [6], the authors have stereotyped the metaclass *Component* to represent a connector in Acme. In [8] the authors discuss the convenience of stereotyping the metaclass *Class* (with <<ArchitecturalConnector>>) to represent software connectors. In [19], the authors have defined an UML 2.0 profile to describe the static view of the C3 architectural style (a simplification of the C2 style). But for the C2 style, the problems previously mentioned prevent any stereotype of *Connector* from representing a C2 connector.

In spite of we can represent a C2 connector stereotyping the metaclass *Component,* we think that UML must have capabilities to represent architectural elements without stereotypes as it is indicated in [13]. From our point of view, it would be necessary to define a new metaclass in UML 2.0 to characterize an architectural connector. This new metaclass could be named *ArchConnector*. This metaclass must be able to define both state and behavior. For example, in the case of a pipe, the behavior would refer the operations of reading and writing in a pipe (with the synchronization conditions for full and empty pipe) while the state would refer to data contained in the pipe at that moment. To do this, *ArchConnector* have to inherit from *Class* (as *Component*), thus a connector should have attributes and operations and take part in associations and generalizations. Since the metaclass *Class* is a subtype of *EncapsuledClassifier*, a connector can have an internal structure and a set f ports to formalize its interaction points. After defining a generic architectural connector, to define specific connectors (as a pipe or a C2 connector) we would specialize the metaclass *ArchConnector*. For example, in the case of a C2 connector, the metaclass to represent it must have defined two operations: *processRequest* and *processNotification* and the state would be defined by the messages queues associated to every port.

5 Conclusions and Future Work

In this paper we have shown that the UML 2.0 metamodel cannot represent software connectors whether we use the metaclass *Connector*. The semantics of connectors in UML 2.0 cannot indicate anything additional to the elements that they connect. UML 2.0 does not permit to characterize the behavior of a connector: buffering and synchronization in the case of pipes and filtering and routing policies in the case of

the C2 connectors. Moreover, in the last style, a connector of UML 2.0 can not be connected to other connectors; it cannot be a composite element nor taking part of any composition. The core problem is that the metaclass *Connector* is not a type of *Classifier* as *Component* is. So, connectors in UML 2.0 do not appear to be first class entities, against the opinion of the software architecture community. Furthermore, the stereotyping mechanism of UML 2.0 allows representing an architectural connector by stereotyping metaclasses as *Component*. However, UML 2.0 Superstructure RFP already indicated that the ability to model architectures should be part of the core modeling capabilities of UML rather than being limited to a profile.

Therefore, UML 2.0 presents shortcomings to describe software architectures (which was one of the goals of the new language version). From our point of view, it would be necessary to define a new constructor in UML 2.0 to characterize an architectural connector. This would imply the definition of a new member in the UML family of languages.

In the short term, our research will focus on the different architectural styles defined nowadays with the purpose of capturing their similarities and establish a set of metaconstructors needed add to UML 2.0 to be able to represent them. The proposed extensions, derived from that study, will define a new member in the UML family of languages that could be named UML-Arch. This set of extensions would be sent to the corresponding RTF (Revision Task Force) of OMG so that they could be considered in the next language review.

References

1. Allen, R.: A formal approach to software architecture. (Doctoral Dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA.). (1997).
2. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J.: "Documenting software architectures, views and beyond". Addison-Wesley/Boston, Massachusetts (2003).
3. Douglass, B.P: "UML 2.0 incrementally improves scalability and architecture". http://www.elecdesign.com/Articles. (2003).
4. Garlan, D. and Kompanek, A.J.: "Reconciling the needs of architectural description with object-modeling notation"; UML 2000 – The Unified Modeling Language: Advancing the Standard. Third International Conference. Springer-Verlag/York, UK (2000).
5. Gomaa, H. and Wijesekera D.: "The role of UML, OCL and ADLs in software architecture"; Proc. of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering, Toronto, Canada (2001).
6. Goulão, M. and Brito e Abreu, F.: "Bridging the gap between Acme and UML 2.0 for CBD"; Proc. of Specification and Verification of Component-Based Systems (SAVCBS'03), workshop at ESEC/FSE (2003).
7. Kandé, M. M. and Strohmeier, A.: "Towards a UML profile for software architecture descriptions"; UML 2000 – The Unified Modeling Language: Advancing the Standard. Third International Conference, Springer-Verlag/York, UK (2000).
8. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B. and Oviedo Silva, J.R. (2004): "Documenting architectural connectors with UML 2". Workshop on Software Architecture Decription & UML, Seventh International Conference on UML Modeling Languages and Applications. Lisbon, Portugal (2004).

9. Lüer, C. and Rosenblum, D.S.: "UML component diagrams and software architecture-experiences from the WREN project"; Proc. of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering, Toronto, Canada (2001).
10. Magee, J. y Kramer, J. (1996). Dynamic structure in software architectures. In Proceedings of SIGSOFT'96: The Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, (pp. 3-14).
11. Medvidovic, N. "Architecture-based specification-time software evolution", Doctoral Dissertation, University of California, Irvine (1999).
12. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F. and Robbins, J.E.: "Modeling software architectures in the unified modeling language"; ACM Transactions on Software Engineering and Methodology, 11, 1 (2002), 2-57.
13. Object Management Group: "Request for proposal: UML 2.0 superstructure RFP." (2000).
14. Object Management Group: "Unified Modeling Language specification" (version 1.4), (2001).
15. Object Management Group: "UML 2.0 OCL Specification (ptc / 03-10-14) (2003).
16. Object Management Group: "UML 2.0 Diagram Interchange Specification (ptc/03-09-01)". (2003).
17. Object Management Group, "UML 2.0 Infrastructure Specification (ptc/03-09-15)". (2003).
18. Object Management Group: "UML 2.0 Superstructure Specification (ptc/04-10-02)". (2004).
19. Pérez-Martínez, J.E. and Sierra-Alonso, A.: "UML 1.4 versus UML 2.0 as languages as to describe software architectures"; Proc. of the First European Workshop on Software Architectures (EWSA'04), St. Andrews, Scotland (2004).
20. Rausch, A.: "Towards a software architecture specification language based on UML and OCL"; Proc. of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering, Toronto, Canada (2001).
21. Riva, C., Xu, J. and Maccari, A.: "Architecting and reverse architecting in UML"; Proc. of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering, Toronto, Canada (2001).
22. Selic, B.: "On modeling architectural structures with UML"; Proc. of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering, Toronto, Canada (2001).
23. Shaw, M., DeLine, R. y Zelesnik, G.: "Abstractions and implementations for architectural connections"; In Proceedings of 3rd International Conference on Configurable Distributed Systems. Annapolis, Maryland (1996).

Semantic Validation of XML Data – A Metamodeling Approach

Dan CHIOREAN, Maria BORTES, Dyan CORUTIU

Babes-Bolyai University, Computer Science Research Laboratory
Str. M. Kogalniceanu, 1
400084 Cluj-Napoca, Romania
http://lci.cs.ubbcluj.ro/ocle
chiorean@cs.ubbcluj.ro

Abstract. The growing usage of the XML standard for information interchange imposes syntactic and semantic correctness of XML data. Presently, DTD and XML Schema offer support mainly for syntactic validation of XML documents. Semantic validation remains an open issue, as several proposed techniques, based on XSL, prove to be limited in expressing semantic rules. The classic way to perform semantic validation has the drawback of being highly coupled with the application logic or with the data representation format. Using the proposed approach, it is possible to uniformly describe with the same formalism - UML – data extracted from various sources, including XML documents, and application logic concepts. Consequently, a single formalism – OCL – can be employed to specify semantic validation rules for structured data and complex rules for integrating data into application logic. Data semantic consistency can be achieved by checking the consistency of a UML model against a set of validation rules specified in OCL. For each DTD or XML Schema, a UML model, semantically equivalent with the described structure, can be automatically constructed. The UML model will serve as a repository for storing the XML information as instances of this model. Different OCL constraints can be specified, both at the metamodel level and model level. Semantic validation of XML data is achieved by checking the UML model against these constraints. **Keywords**: syntactic validation, semantic validation, XML documents, DTD, XML Schema, reverse engineering DTD, UML, OCL, checking UML model, XMI support, AOP.

1 The case for XML semantic validation

eXtensible Markup Language (XML) is a standard interchange format [1] widely used in the Web environment. To achieve XML information interchange, the applications shall agree on the structure of the exchanged XML documents. The structure of XML data is described using definition documents such as DTD [1], XML Schema [2], Relax NG [3], etc. Validation of XML documents against their definitions ensures their syntactical correctness. Most XML parsers perform syntactic validation of XML data, rejecting an invalid XML document.

Currently, XML information exchanged between applications is increasingly complex. Moreover, the applications need to integrate heterogeneous information, stored in different formats [10]. In this context, syntactic validation of XML data becomes

insufficient. To ensure integration and interoperability, XML data needs to conform to complex semantic constraints:

- relations between information located in the same XML document or in different XML documents,
- relations between information located in a XML document and information stored in a database,
- relations between information located in a XML document and application specific information,
- value constraints for attribute values or for element content.

The rest of the paper is structured in six sections. Section 2 analyses the State of the art in the XML semantic validation domain. The drawbacks of current approaches are captured in Section 3. The next section - 4, describes the concept, methodology and tool support for the proposed metamodeling approach. In Section 5, the advantages of our approach are briefly summarized. An example that applies the metamodeling approach in realizing XML semantic validation in a multi-tier application is presented in Section 6. The last section - 7 contains the conclusions and remarks on the future work. References and two appendices presenting the stereotypes defined in the UML DTD profile, and the DTD files used in the example were included at the end of the paper.

2 State of the art

In the classical approach, the concern of XML data semantic validation is addressed at the application level. The application is a tolerant system that accepts any information extracted from syntactically valid XML documents. When data received does not fulfill application preconditions, we obtain undesired application behavior.

The main disadvantage of the classical approach is the high coupling between application logic code and semantic validation code. The slightest modification of semantic rules that apply to information extracted from XML documents implies changes in the application code, recompilation, redeployment, and then restarting the application. These operations are not acceptable for critical systems.

The modern techniques for checking XML data semantic validity avoid the above-mentioned drawback through a declarative approach, decoupled from the application logic. Most of these techniques are based on XSL technology.

Schematron [6] is an XPath based constraint language “for making assertions about patterns found in XML documents”. It supports the expression of assertions between arbitrary elements in XML documents, selected with paths. The violation of the assertions is signaled through a custom error message that can be formatted as a XML document. Semantic validation with Schematron implies the translation of assertions into a validating XSL stylesheet, processed by a XSLT processor, or the evaluation of

assertions by a specific evaluation engine. This approach reached a maturity level and, at this moment, follows a standardization process as ISO Schematron.

xLinkit [7] is a constraint language based on XPath and first order logic. xLinkit allows the expression of complex inter-document constraints, being used for realizing semantic validation of large and complex XML documents: financial business documents, XMI [12] documents, etc. The combination between paths and first order logic in xLinkit offers an increased power of expression for XML data semantic constraints.

XML Constraint Specification Language (XCSL) [9], is a XML based domain specific language with the purpose of allowing XML designers to restrict the content of XML documents. It is a simple, and small language tailored to write contextual conditions constraining the textual value of XML elements in concrete documents. An XCSL document, describing the constraints to be validated, is given to the XCSL Processor Generator that produces a XSL stylesheet; then, using any standard XSL processor, it is possible to apply that stylesheet to the XML document needed to be checked. The obtained result is another XML document with the error messages. OASIS' Content Assembly Mechanism (CAM) [8], provides an open XML based system for using business rules to define, validate and compose specific business documents from generalized schema elements and structures. A CAM rule set and document assembly template defines the specific business context, content requirement, and transactional function of a document. A CAM template must be capable of consistently reproducing documents that can successfully carry out the specific transactional function they were designed for. CAM also provides the foundation for creating industry libraries and dictionaries of schema elements and business document structures to support business process needs.

3 Drawbacks of current approaches in XML validation

The formalisms used in XML validation that rely on XSL and XPath, are difficult to write, understand and maintain. For example, the well formedness rule in the UML language that specifies the uniqueness of the association end names of an association is specified in the OCL language as:

```
context Association
inv:
   self.connection->forAll(c1, c2 | c1.name=c2.name implies
   c1=c2)
   -- an equivalent, but more compact specification is:
   -- self.connection->isUnique(c | c.name)
```

The same rule expressed in xLinkit [7] formalism is:

```
<forall var="a" in="$associations">
   <forall var="x"
    in = "$a/Foundation.Core.Association.connection/
          Foundation.Core.AssociationEnd">
      <forall var="y"
       in="$a/Foundation.Core.Association.connection/
       Foundation.Core.AssociationEnd">
      <implies>
      <equal op1="$x/Foundation.Core.ModelElement.name/text()"
           op2="$y/Foundation.Core.ModelElement.name/text()"/>
      <same op1="$x" op2="$y"/>
      </implies>
      </forall>
   </forall>
</forall>
```

The xLinkit rule applies to an XML document containing a UML model in XMI format. A DOM tree is created for the XML document and the navigation through XML elements is realized with paths. In complex documents, a rule that applies to deep elements implies the specification of deep paths, making the assertion expression less readable. Assertion comprehensiveness is increased using relative paths, but this leads to a decrease in performance, as the relative paths are searched in the whole DOM tree.

Another drawback is related to the results of the XML data validation. Most approaches have as an output an error report that can be consulted to identify where validation errors occurred. If we deal with an important XML document, in which there is valuable information, we would like to recover the document and make it compliant with validation rules. The report marks the errors and their location; only using an XML editing tool the recovery process can be performed. Even in cases of dedicated XML editors, large XML documents recovery is a complicated and tedious process. Today, document recovery is a very common and important problem for UML models saved in XMI format. Most XMI parsers do not fully comply with the XMI version for which they were built. Therefore, in many cases, XMI parsers fail in importing the models exported by other tools. It would be very useful to recover as much information as possible from such a document in order to facilitate a complete model interchange.

Finally, the most complex and important problem considers the semantic validation of XML data in the context of data integration in multi-tier applications. In such applications we can distinguish several categories of semantic rules, including: rules for data consistency, business rules, integration rules, etc. Due to their strong relation with the XML standards family, the XSL and XPath based approaches are not feasible in specifying integration rules between different application layers.

4 A metamodeling approach for XML validation

As mentioned in the previous section, several categories of rules can be specified in the context of an application. Our intent is to simplify and unify the process of specification and validation of rules, at any level in a system, by means of the UML/OCL standard. UML is primarily targeted at modeling applications. The same formalism – UML – is also capable to define several domain specific languages, such as DTD or XML Schema, through its lightweight extension mechanisms. The advantage of UML representation is that it straightforwardly allows specification of semantic rules with OCL – a textual formalism for UML. Thus, XML semantic validation can be reduced to UML model validation against OCL semantic rules. This approach has lower costs in the development process since it employs the intensive usage of a single standard – UML – for the specification and validation of any rule in an application.

The metamodeling approach for XML validation implies conceptual, methodological and tool support.

From a conceptual point of view, the problem of XML semantic validation translates into a UML model consistency problem. DTD and XML Schema are mechanisms for metadata definition. A XML document contains instances of metadata described in the associated definition documents. A UML model is constructed for the metadata from DTD and XML Schema documents. This model is instantiated with the information present in XML.

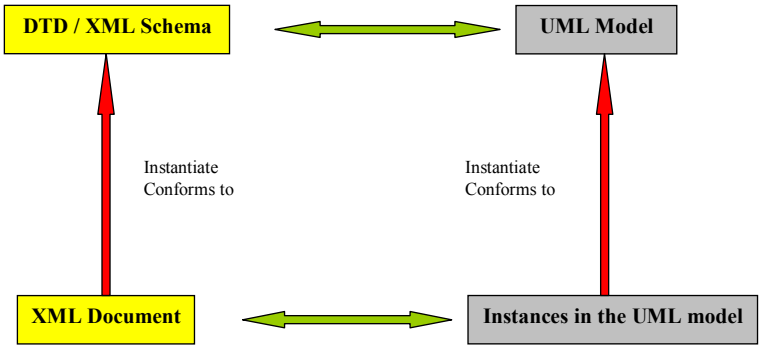


Figure 1 The correspondence between XML data validation and UML model valid

For example, the UML model presented in Figure 3 is a model that corresponds to the following DTD:

```
<!ELEMENT Transaction (Client, Account, Date, Amount)>
<!ELEMENT Client EMPTY>
<!ELEMENT Account EMPTY>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Amount EMPTY>
<!ATTLIST Client id ID #REQUIRED>
<!ATTLIST Account id ID #REQUIRED>
<!ATTLIST Amount value CDATA #REQUIRED>
```

Figure 2 A DTD describing the structure of a transaction document

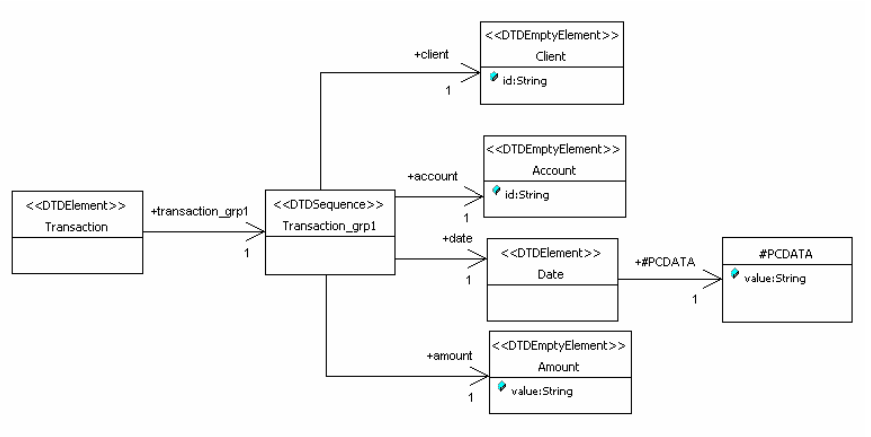


Figure 3 A UML model complying with the structure described in Figure 2

In order to build a semantically equivalent UML model for the document definitions, we must associate additional semantic information with UML model elements. This can be achieved by means of a UML profile for DTD or XML Schema. This profile employs lightweight extension mechanisms like stereotypes, tagged values and constraints in order to express specific semantics associated to DTD or XML Schema concepts.

In order to guide users in applying this approach, a very simple process (presented in Figure 4) was conceived. Each activity will be described in detail in the following subsections.

Regarding the tool support, the proposed approach was implemented in OCLE[1] in order to benefit from the tool's OCL specification and evaluation features. This tool supports: the automatic construction of UML models (by a reverse engineering process on DTD documents), the automatic construction of snapshots that instantiate the model, by parsing XML documents, a natural and intuitive model navigation by means of OCL, error rationale identification and XML document recovery.

[1] Acronym for Object Constraint Language Environments tool, conceived and implemented at LCI, see http://lci.cs.ubbcluj.ro/ocle

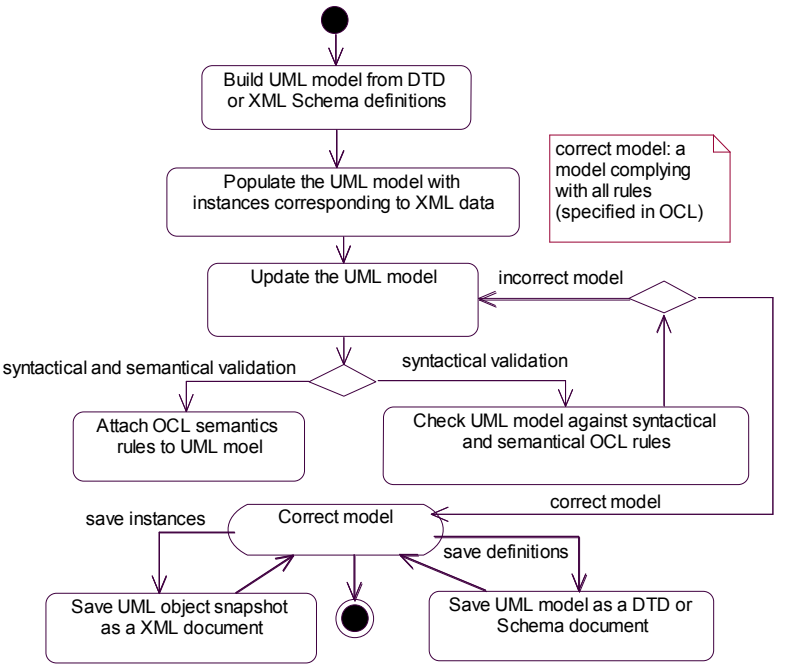


Figure 4 Activities of the XML validation process

4.1 Building a UML model from data definitions

The UML model presented in Figure 3 was automatically constructed from the DTD definitions using OCLE. The class diagram was realized by drag and dropping the elements from the model browser. Comparing the diagram and the DTD representing the input, we notice that:

- Classes contained in the class diagram can be grouped in two categories: one corresponding to DTD elements (`Transaction`, `Client`, `Account`, `Date`, `Amount`), another containing classes that model the DTD containment relationships between elements (`Transaction_grp1` and `#PCDATA`).
- Excepting the class named `#PCDATA` (that models DTD text elements), all other classes are stereotyped in order to preserve their specific DTD semantic. The semantics of each stereotype in the DTD profile is described in Annex A.

- The containment relationships between DTD elements are modeled as UML unidirectional associations, from the container to the contained elements. The DTD element multiplicity is mapped to the corresponding association end multiplicity.
- A tagged value is attached to the association ends corresponding to the contained elements of container classes (e.g. classes stereotyped with `<<DTDSequence>>`). This value represents the position of that element in the sequence.
- Attributes of DTD elements are modeled as attributes defined in the corresponding classes, having the same name. Attributes are also stereotyped in order to preserve additional DTD information such as `REQUIRED` or `FIXED`. In our example, the Client and Account attributes, both named id, are stereotyped as `<<REQUIRED>>`.

The UML model created in the DTD reverse engineering process can be "refined" by changing the type of class attributes. For example, in the model represented in Figure 3, the type of `Amount`'s value attribute can be changed from `String` to `Integer` if the new type is better suited to user's requirements.

4.2 Populating UML model with instances corresponding to XML data

Once the UML model is created, it can be populated with instances corresponding to XML data. The OCLE tool supports this operation in an automated manner. For each XML construct, an instance of the appropriate UML model element (represented in Figure 3) is created. The UML metamodel specifies that all instances be contained in a `Collaboration` object. Therefore, before populating the model with instances, a `Collaboration` object needs to be created. The data contained in a XML document is imported as UML model instances into the `Collaboration` object.

For example, by importing the following XML document, several UML instances are created, as shown in Figure 6:

```
<?xml version="1.0" encoding="UTF-8"?>
<Transaction>
<Client id="122334444556"/>
<Account id="RO144432323335667"/>
<Date> 09-06-05 </Date>
<Amount value="23000"/>
</Transaction>
```

Figure 5 XML data complying with DTD represented in Figure 2

In order to create the snapshot represented in Figure 6 using OCLE, after "importing" objects from XML data, an object diagram needs to be created. Dragging and dropping the newly created objects from the model browser in the object diagram, the objects and the relationships among them are drawn automatically.

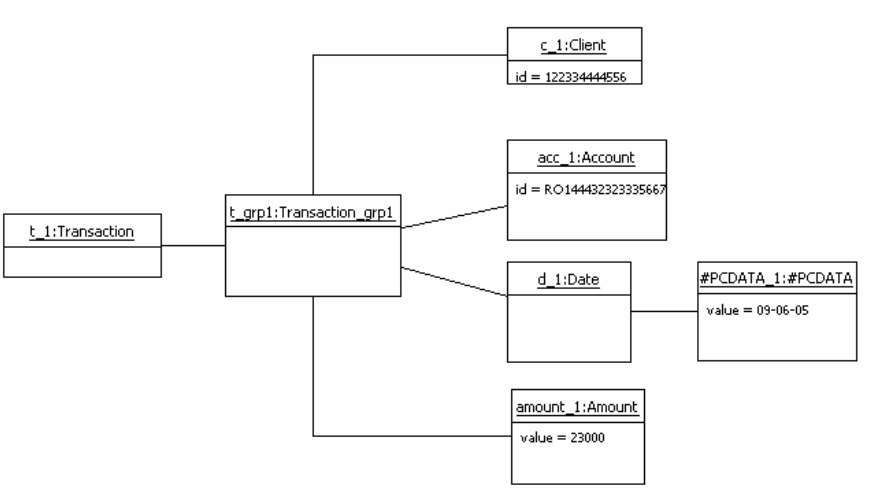


Figure 6 - The snapshot created after importing XML data represented in Figure 5

We can illustrate the mapping rules between XML elements and UML instances by comparing the XML document with the above snapshot diagram:

- For each XML element an UML `Object` is created as an instance of the UML `Class` that corresponds to the DTD definition of that XML element.
- The UML `Object` created for a XML element contains the slots (`AttributeLinks`) corresponding to the DTD attributes specified for that XML element.
- The containment relationship between XML elements is mapped to a UML `Link` that instantiates the association that corresponds to the DTD containment relationship between the definitions of the XML elements.

Any unexpected XML element (e.g. an element or attribute not defined in DTD, or an element that occurs in an unexpected location) will be processed in the following manner:

- a UML `Class` for that element will be created, marked with the stereotype `<<DTDUndefined…>>`,
- a corresponding UML `Object` will be created as an instance of the above-mentioned `Class` and marked with the stereotype `<<XMLUnexpected…>>`,
- a UML `Association` will be created between the `Class` corresponding to the XML unexpected element and the `Class` corresponding to the XML container element, marked with the stereotype `<<DTDUndefined…>>`,
- a `Link` will be created as instance of the above-mentioned `Association`, also marked with the stereotype `<<XMLUnexpected…>>`.

The processing of an unexpected element implies appropriate changes in the UML model, aiming to preserve model consistency. The newly introduced UML structural

model elements will be stereotyped as <<DTDUndefined…>>. In a similar manner, the absence of a required element will cause the instantiation of the missing element. This instance will be marked as <<XMLMissing…>>. The basic idea of this approach is to construct an evolving UML model, without losing XML information. It is the user who decides whether or not to keep an UML model element stereotyped as <<Undefined>>, <<Unexpected>> or <<Missing>>.

The graphical representation is more comprehensive than the textual representation. With the help of diagrams, objects for which constraint evaluation failed can be isolated, thus allowing easy identification of error rationale. Once the error rationale is identified, the user can perform document recovery by modifying the corresponding model with the purpose of obtaining a valid UML model.

4.3 Checking the UML model against rules specified in the UML DTD profile

The UML DTD profile contains stereotypes, constraints and tagged values. OCL constraints are defined at the UML metamodel level, corresponding to the informal constraints stated in W3C XML1.0 specification [1]. Some structural rules imposed by DTD have an equivalent specification in UML Well Formedness Rules (WFR) [11].

For example, in the context of a DTD choice group (see [1]), the following informal rule is defined: "*Any content particle in a choice list MAY appear in the element content at the location where the choice list appears in the grammar*". In the UML DTD profile, this rule is modeled as an exclusive OR between all associations of a container with its contained elements:

```
context Object
   inv DTDChoice:
      if self.classifier->forAll(c | c.stereotype->exists(s |
         Set{'DTDChoice'}->includes(s.name)))
         then self.allOppositeLinkEnds->collect( lE: LinkEnd|
            lE.associationEnd)->select(ae| ae.isNavigable and
            ae.participant.hasDTDStereotype())->size = 1
         else true
       endif
```

The operation hasDTDStereotype() defined in the Classifier context, is specified below.

```
context Classifier
   def:
      let hasDTDStereotype(): Boolean =
         if self.stereotype->exists(s | Set{'DTDElement',
            'DTDEmptyElement', 'DTDAnyElement', 'DTDSequence',
      'DTDChoice', 'DTDMixed','DTDAny', 'DTDUndefinedElement'}
            ->includes(s.name))
            then true
            else false
         endif
```

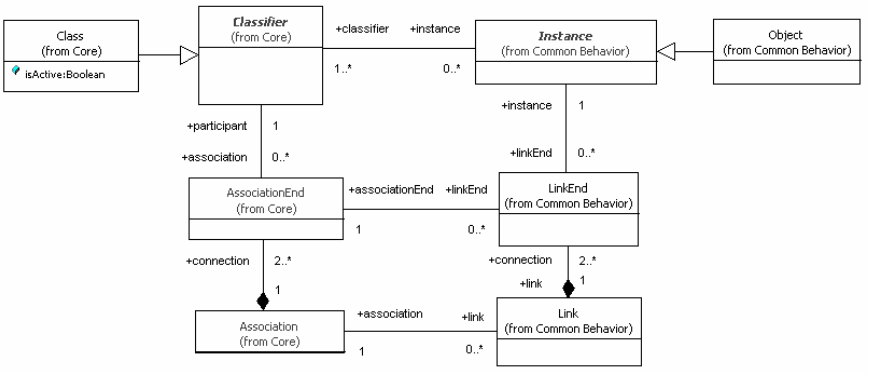


Figure 7 – The UML 1.5 metamodel elements implied in the DTDChoice invariant

The operation `hasDTDStereotype()` defined in the `Classifier` context, is specified below.

```
context Classifier
   def:
      let hasDTDStereotype(): Boolean =
         if self.stereotype->exists(s | Set{'DTDElement',
            'DTDEmptyElement', 'DTDAnyElement', 'DTDSequence',
      'DTDChoice', 'DTDMixed','DTDAny', 'DTDUndefinedElement'}
            ->includes(s.name))
            then true
            else false
         endif
```

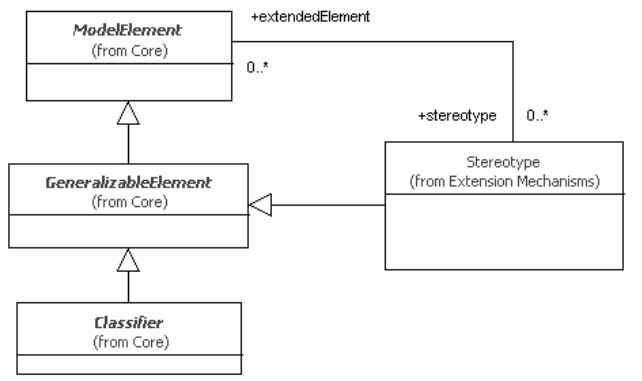


Figure 8 - Modeling Stereotype in the UML 1.5 metamodel

Attributes defined in DTD must also conform to different rules. The following W3C validity constraint: "*If the default declaration is the keyword #REQUIRED, then the attribute MUST be specified for all elements of the type in the attribute-list declaration.*" was specified in the UML DTD profile, as shown below:

```
context Attribute
   inv REQUIRED:
      if self.stereotype.name->includes('REQUIRED')
         then self.attributeLink.value->any(v |
              v.isUndefined)->isEmpty
         else true
      endif
```

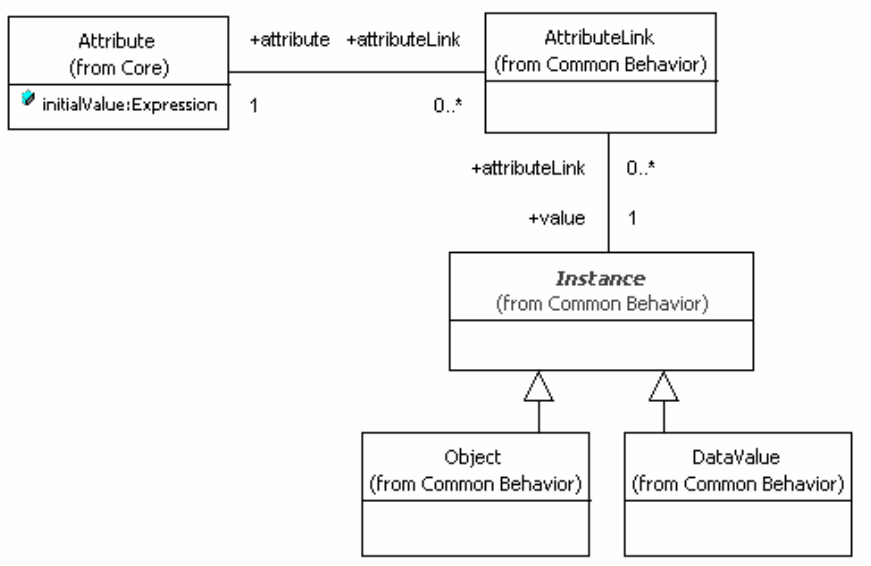


Figure 9 - The UML 1.5 metamodel elements implied in the REQUIRED invariant

The UML DTD profile provides OCL constraints that specify syntactical rules for XML content. The syntactical validation of XML data against DTD definitions is realized through model validation against rules from the UML DTD profile.

4.4 Specifying and evaluating semantic rules associated with the UML model

The metamodeling approach requires that semantic rules be specified at M1 level. At this level, we can provide rules for XML data validation or rules for integration of XML data in the application. The example in Section 6 illustrates the specification of such rules.

Similar to invariants specified at M2 level, the invariants specified at M1 level can be evaluated at M0 level (at runtime, if the OCL specifications are translated into programming language code), or at M1 level if OCL expressions are evaluated.

In case the evaluation process reveals errors, the user can choose to recover the invalid document by modifying the corresponding model. The evaluate-modify sequence of activities is repetitive and ends when the evaluation reports no errors. After successful validation it makes sense to revert the model back to the XML representation and its DTD definition.

5 Benefits of the metamodeling approach

Compared with the approaches mentioned in the State of the Art section, the metamodeling approach offers some advantages:

- The solution does not require a new formalism for specifying semantic rules for XML data, rather it employs a broadly used standard (UML) that successfully addresses all semantic aspects of this particular problem;
- OCL constraints are much more clearer, because OCL is a powerful model navigation language;
- XML semantic validation can be realized within a UML/OCL tool or through runtime execution of programming language generated code for OCL rules;
- The graphical notation of UML provides a more intuitive view for the structure of XML data;
- In the context of a UML/OCL tool, document recovery becomes a simple model updating task and does not require the implementation of a special feature,
- Using modeling techniques at successive abstraction levels enables us to solve the XML semantic validation problem in a more scalable manner.

6 Applying the metamodeling approach – an example

In this example we consider the design of a multi-tier application consisting of at least two tiers: the data tier (containing several XML documents) and the logical tier (describing the application architecture). Our objective is to achieve semantic consistency of information stored in the data tier, in the context of its integration with the logical tier. Both logical and data tiers are represented in UML and the semantic rules are expressed in OCL.

Problem statement: A bank has several branches in different cities. This bank keeps its interest rates in a XML file that conforms to the `InterestRates.dtd`. Each branch manages a set of customer accounts. Customer and account information is stored in XML files conforming to `Customers.dtd`, respectively `Accounts.dtd` (DTD files are shown in the Appendix B).

The UML model shown below describes the logical tier for our example. This tier is not aware about the source of information for interest rates, customers or accounts

(these sources may be relational databases and/or XML files or other data repositories).

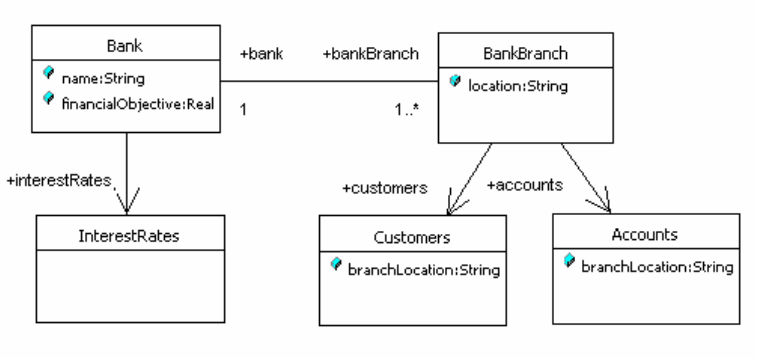


Figure 10 UML model showing the logical tier

After reversing DTD definitions for interest rates, customers and accounts, the model presented in Figure 10 will be modified in order to integrate concepts from both application tiers:

The stereotyped UML classes were found in the DTD definitions. Note that `Accounts`, `Customers` and `InterestRates` concepts appear in both tiers. In this way, the classes corresponding to DTD descriptions can be automatically linked to the logical tier. The mentioned elements connect the logical tier with the data tier. In case these elements were present only in the data tier, the connections between them and the classes from the logical model should be done by hand.

For the UML model obtained after reversing DTD definitions, we provide a set of OCL specifications enabling the semantic validation of XML data. The final goal of this semantic validation is the estimation of the bank's monthly benefit and the comparison of this benefit with the financial objective of that bank for the current year. This comparison can be useful in establishing the bank's short-term strategy in order to achieve its objective.

First, we need to ensure that all XML data comply with the data types expected by the logical layer. For example, the type of the `InterestRate`'s attribute named `type` is `AccountKind`, an instance of the UML Enumeration metaclass. This means that `type`'s values are restricted to the enumeration literals `deposit` and `loan`. The other two attributes of the mentioned class have the type `String`, but they have different semantics. The value of `term` attribute should be a valid `Integer` and `rate` attribute value should be a valid `Real`. In OCL, these constraints can be expressed as:

```
context InterestRate
   inv semanticTypeValidity: (type = #deposit or type = #loan)
       and term.toInteger().oclIsKindOf(Integer) and
           rate.toReal().oclIsKindOf(Real)
```

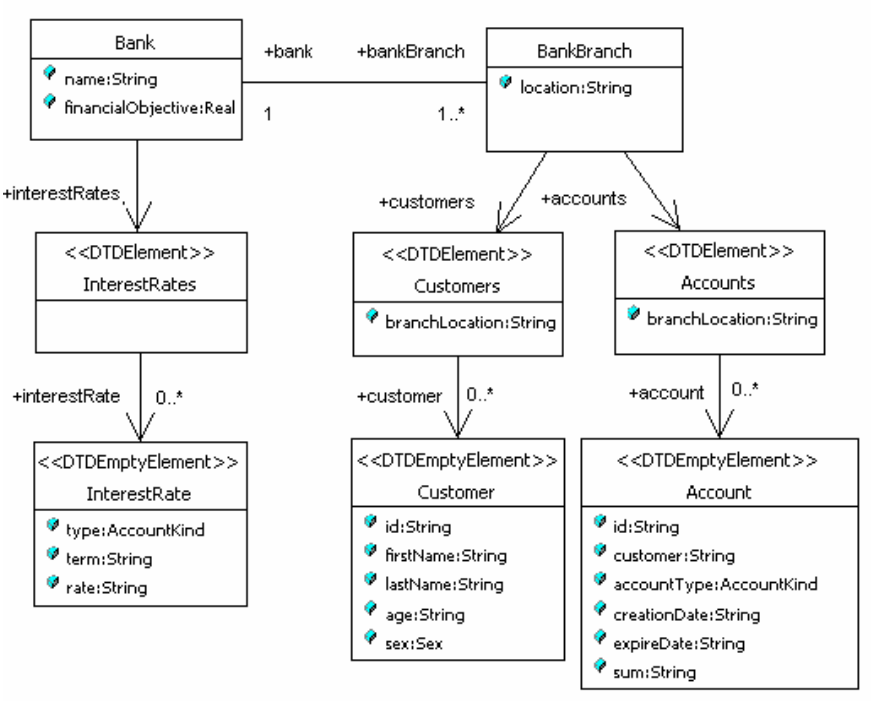


Figure 11 UML model after reversing DTD definitions

Furthermore, OCL can be used to enforce semantic validity of XML data loaded from several XML documents. The invariant below imposes that all accounts managed by a branch belong to customers registered to that branch. This rule restricts the value domain for the `customer` attribute of every `Account` instance managed by a branch to the collection of `id` attribute values of `Customer` instances registered to the same branch:

```
context BranchBank
   def:
-- get a set containing the "id" values for Customer instances
      let customerIds: Set(String) =
          self.customers.customer->collect(c | c.id)->asset

   inv validCustomerValues:
      self.accounts.account->forAll(a | customerIds
           ->includes(a.customer))
```

The next specifications illustrate the semantic relationships between XML data and the UML logical model in which the data was integrated. For example: the location of

a BankBranch should be identical with the branchLocation attribute value of Accounts and Customers instances connected to this branch.

```
context BankBranch
   inv hasSameLocation:
      self.accounts.branchLocation = location and
      self.customers.branchLocation = location
```

Now we return to our final goal: estimation of bank benefit for the current month and comparison of the obtained value with the bank's financial objective. We assume that the bank offers only monthly interest payment. For obtaining the bank benefit for the current month, we have to compute the monthly interest for each account in every branch of the bank.

In the InterestRates context, we can specify an OCL operation that returns the monthly interest rate for a given account type and term:

```
context InterestRates
   def:
      let monthlyInterstRate(accountType:AccountKind,term:
          Integer): Real = self.interestRate->any(r | r.type =
          accountType and r.term.toInteger() =
          term).rate.toReal()/1200
```

This specification can be used to compute the monthly total that a bank branch should receive from or pay to its customers:

```
context BankBranch
   def:
-- navigate and cache the InterestRates instance connected to
-- the bank
      let interestRates: InterestRates =
          self.bank.interestRates
-- estimate the monthly interests for a bank branch
      let estimateMonthlyInterests: Real =
          self.accounts.account->iterate(a: Account; total:
          Real = 0 |
             if (a.accountType = #deposit)
                then
-- for deposits the branch should pay the value of monthly
-- interest rate to its customers
                     total - interestRates.monthlyInterestRate
               (a.accountType, a.periodInMonths)*a.sum.toReal()
                else
-- for loans the branch should receive the value of monthly
-- interest rate from its customers
      total + interestRates.monthlyInterestRate(a.accountType,
      a.periodInMonths)*a.sum.toReal()
            endif)
```

Using the function previously described, we can specify an invariant that estimates whether the monthly bank's benefit has a value that conforms to the financial objective:

```
context Bank
   inv conformsToFinancialObjective:
      self.bankBranch->iterate(b: BankBranch; benefit: Real=0
      | benefit + b.estimateMonthlyInterests) >
      financialObjective/12
```

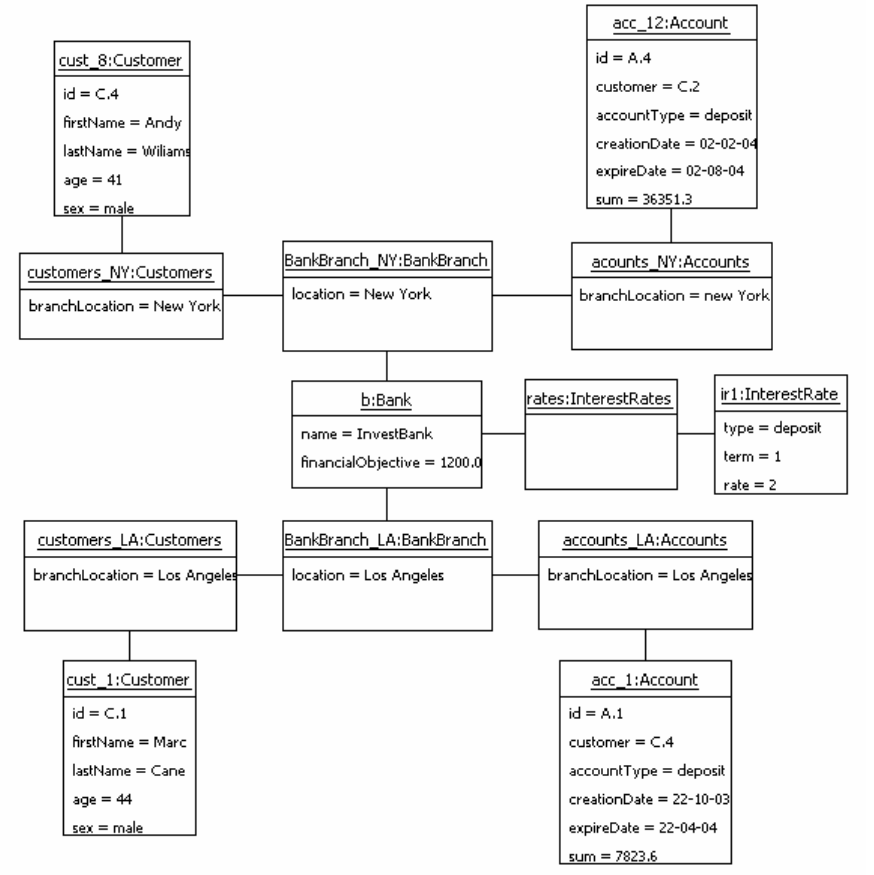


Figure 12 OCLE Snapshot showing some of the instances of Bank model classes

In order to perform a static evaluation of the specified invariants, apart from instances obtained after populating the UML model with XML data, we need to create instances of logical tier classes. Using OCLE instance editor, we create a `Bank` instance, two `BankBranch` instances and the links connecting the bank with each branch. All the other instances represented in Figure 12 were created automatically during XML data

import. The links connecting the logical tier instances with instances corresponding to XML data, are also created using the OCLE instance editor.

By evaluating OCL specifications for the constructed snapshot we can detect and correct XML data inconsistencies.

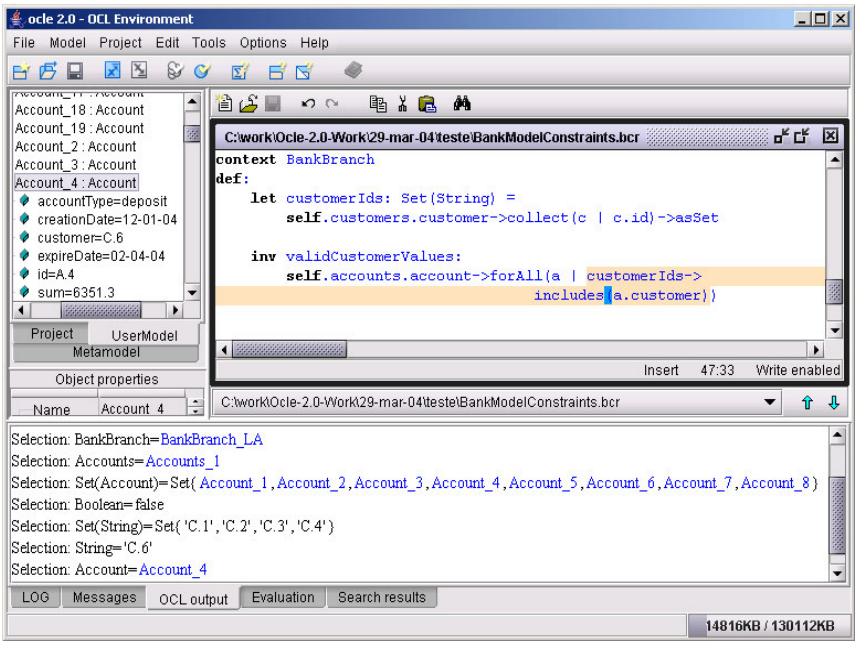


Figure 13 Using OCLE to evaluate subexpression for an invariant that failed

Figure 13 below shows the OCLE capabilities in identifying the rationale of an invariant failure. The `validCustomerValues` invariant failed for `BankBranch_LA` instance. Evaluating the subexpressions of this invariant, we find that customerIds operation returns a set containing the following ids: "`C.1`", "`C.2`", "`C.3`" and "`C.4`". `Account_4` instance has a "`customer`" attribute value of "`C.6`", which designates a customer that is not registered to this branch. The inconsistence can be corrected by modifying the "`customer`" attribute value for `Account_4` instance to contain a valid customer id or by deleting the `Account_4` instance.

7 Conclusions and future work

Validating XML documents using metamodeling approach offers many benefits compared with all existent approaches. The price to pay for the benefits obtained using this approach is cheap because it's about a larger memory resource, a requirement easy to satisfy.

The proposed solution is based on raising the abstraction level and on using a richer formalism compared with those used for data representation. Thus, representation mapping from DTD or XML Schema to a UML model and from XML data to instances of previously created UML model elements is realized without losing information. The obtained UML model can be enriched in order to support more complex and detailed validations. The formalism used (UML + OCL) is representation independent, more suggestive, (due to the graphical representation) and easier to understand (due to a higher abstraction level) [14] [16].

The proposed process also offers other advantages. XML data validation can be realized at design time supporting users in specifying application logic; the process supports XML data recovering; rule specification for architectures of multi-tier applications can be done in an unified manner. Runtime data validation is supported through execution of automatically generated code from OCL specifications. Mapping OCL assertions in aspects specified in programming languages is the preferred solution in many cases. In these cases, the application code used in validation becomes an aspect completely separated from the application logic.

Due to the above mentioned features, the approach seems to be very well suited and recommended for validating and querying Web Semantics documents. In order to do this, the following future work is planned:

- Conceiving and implementing a MOF based OCL tool,
- Specifying XML Schema profiles, RDF and OWL profiles; designing and implementing Reverse Engineering functionalities for the above mentioned standards,
- Extending the OCL support in order to support model mapping specification and model execution,
- Extending the OCL and improving the code generator in order to support aspect specifications in OCL and their mapping to programming languages.

References

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler; *Extensible Markup Language. W3C Recommendation*; World Wide Web Consortium; Oct. 2000; http://www.w3.org/TR/2000/REC-xml-20001006

[2] D. C. Fallside, Priscilla Walmsley; *XML Schema Part 0: Primer Second Edition. W3C Recommendation*; World Wide Web Consortium; Oct. 2004; http://www.w3.org/TR/xmlschema-0/

[3] J. Clark, M. Murata; *RELAX NG Specification. Committee Specification*; Organization for the Advancement of Structured Information Standards (OASIS); Dec. 2001; http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[4] J. Clark; *XSL Transformations (XSLT). Technical Report*; World Wide Web Consortium; Nov. 1999; http://www.w3.org/TR/xslt

[5] J. Pollack; *Syntactic and Semantic Validation within a Metadata Management System*, presented at the EO/GEO Meeting Fredericton, NB, Canada, June 25-29, 2001; http://gcmd.gsfc.nasa.gov/Aboutus/presentations/conferences/eogeo01/eogeo_01.html

[6] Rick Jelliffe; *The Schematron Assertion Language 1.5. Specification*; Academia Sinica Computing Centre; Oct. 2002; http://xml.ascc.net/resource/schematron/Schematron2000.html

[7] C. Nentwich, L. Capra, W. Emmerich, A. Finkelstein; *xLinkit: a Consistency Checking and Smart Link Generation Service*; ACM Transactions on Internet Technology, 2(2): 151–185, May 2002; http://xml.coverpages.org/xlinkitwp200102.pdf

[8] OASIS Content Assembly Mechanism Technical Committee; *Content Assembly Mechanism (CAM) Specification Document. Committee Draft Version 1.0*; Organization for the Advancement of Structured Information Standards; March 2004; http://www.oasis-open.org/committees/download.php/5914/OASIS-CAM-Specifications-1_0-RC-017C-021904.doc

[9] Marta H. Jacinto, Giovani R. Librelotto, Jose C. Ramalho and Pedro R. Henriques; *XCSL: XML Constraint Specification Language*; May 14, 2004; http://www.di.uminho.pt/~jcr/PROJS/xcsl-www/

[10] Matthias Ferdinand, Christian Zirpins, D. Trastour; *Lifting XML Schema to OWL*; Published in *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings*; Springer Heidelberg; 2004; pp. 354-358; http://vsis-www.informatik.uni-hamburg.de/publications/view.php/204

[11] Object Management Group; *Unified Modeling Language Specification, Version 1.5*; March 2003; http://www.omg.org/cgi-bin/doc?formal/03-03-01

[12] Object Management Group; *XML Metadata Interchange (XMI) Specification, Version 1.2*; Jan. 2002; http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[13] V. Raatikka, E. Hyvönen; *Ontology-Based Semantic Metadata Validation*; In *Towards the Semantic Web and Web Services*, in Proceedings of XML Finland 2002 Conference; pp. 28-40; http://www.cs.helsinki.fi/u/eahyvone/publications/Validation.pdf

[14] Dan Chiorean, Adrian Carcu, Mihai Pasca, Cristian Botiza; *Ensuring UML model consistency using the OCL environment*; Electronic Notes in Theoretical Computer Science – ENTCS/157; 2004

[15] Dan Chiorean, Maria Bortes, Dyan Corutiu; *UML/OCL tools – Objectives, Requirements, State of the Art – The OCLE Experience*, in Proceedings of 11th Nordic Workshop on Programming and Software Development Tools and Techniques; pp. 163- 180 – ISBN 952-12-1385-X, ISSN 1239-1905; 2004

[16] Dan Chiorean, Dyan Corutiu, Maria Bortes; *Good practices for creating correct, clear and efficient OCL specifications*, in Proceedings of 2nd Nordic Workshop on the Unified Modeling Languages pp. 127-142 – ISBN 952-12-1386-8, ISSN 1239-1905; 2004

Appendix: A

UML base class	Stereotype	DTD stereotype semantic (mappings with DTD-XML concepts)
Classifier	DTDChoice	content structure of type “choice”
Classifier	DTDSequence	content structure of type “sequence”
Classifier	DTDMixed	content structure of type “mixed”
Classifier	DTDAny	content structure of type “any”

Classifier	DTDAnyElement	ANY element
Classifier	DTDEmptyElement	EMPTY element
Classifier	DTDElement	generic DTD element
Attribute	CDATA	CDATA type for DTD attribute
Attribute	ID	ID type for DTD attribute
Attribute	IDREF	IDREF type for DTD attribute
Attribute	IDREFS	IDREFS type for DTD attribute
Attribute	NMTOKEN	NMTOKEN type for DTD attribute
Attribute	NMTOKENS	NMTOKENS type for DTD attribute
Attribute	ENTITY	ENTITY type for DTD attribute
Attribute	ENTITIES	ENTITIES type for DTD attribute
Attribute	NOTATION	NOTATION type for DTD attribute
Attribute	REQUIRED	#REQUIRED DTD attribute
Attribute	FIXED	#FIXED DTD attribute value
Classifier	DTDUndefinedElement	element not present in DTD definitions, but introduced in UML model due to occurrence of an undefined tag in XML
Object	XMLUnexpectedElement	instance of a class from the static UML model; reflects the occurrence of a unexpected XML tag relative to a given context (the containing XML tag)
Object	XMLMissingElement	instance of a class from the static UML model; reflects the absence of an expected XML tag relative to a given context
Attribute	DTDUndefinedAttribute	attribute not present in DTD definitions, but introduced in UML model due to occurrence of an undefined attribute for a given XML tag
AttributLink	XMLMissingAttribute	slot that corresponds to a DTD #REQUIRED attribute; reflects the absence of an expected attribute for a given XML tag
AttributLink	XMLUnexpectedAttribute	slot that is an instance of an undefined attribute (not present in DTD definitions); reflects the occurrence of the undefined attribute for a given XML tag
Association	DTDUndefinedAssociation	association that does not exist between classes of the UML model, but introduced in UML model due to occurrence of a unexpected XML tag relative to a given context (the containing

		XML tag)
Link	XMLMissingLink	instance of an association that exists in UML model; reflects the absence of an expected XML tag relative to a given context
Link	XMLUnexpectedLink	instance of an association that was undefined in UML model; reflects the absence of an expected XML tag relative to a given context

Note: The DTD `#PCDATA` is mapped to a UML Class named `#PCDATA`, having an attribute named `value`, that holds the textual value. Also, in order to simulate an `any` content, we use a `mixed` content which contains all the elements defined in that DTD.

Appendix: B

```
Accounts.dtd

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Accounts (Account*)>
<!ELEMENT Account EMPTY>
<!ATTLIST Accounts
branchLocation CDATA #REQUIRED>
<!ATTLIST Account
id ID #REQUIRED
customer CDATA #REQUIRED
accountType (deposit | loan) #REQUIRED
creationDate CDATA #REQUIRED
expireDate CDATA #REQUIRED
sum CDATA #REQUIRED>

Customers.dtd

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Customers (Customer*)>
<!ELEMENT Customer EMPTY>
<!ATTLIST Customers
branchLocation CDATA #REQUIRED>
<!ATTLIST Customer
id ID #REQUIRED
firstName CDATA #REQUIRED
lastName CDATA #REQUIRED
age CDATA #IMPLIED
sex (male | female) #IMPLIED>

InterestRates.dtd

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT InterestRates (InterestRate*)>
<!ELEMENT InterestRate EMPTY>
<!ATTLIST InterestRate
type (deposit | loan) #REQUIRED
term CDATA #REQUIRED
rate CDATA #REQUIRED>
```

Accessibility testing XHTML documents using UML

Terje Gjøsæter, Jan P. Nytun, Andreas Prinz, Merete S. Tveit

Agder University College
Grooseveien 36
Grimstad. Norway
{terje.gjosater, jan.p.nytun, andreas.prinz, merete.s.tveit}@hia.no

Abstract. This paper handles modeling and test of accessibility requirements for web documents. We propose to use metamodelling with UML and OCL for this task. Our own environment within the SMILE project has proposed a basic representation that can be used on all levels in a metamodelling architecture; this representation is used for representation of the elements of the metamodel, models and model instances. We show the use of OCL formulas to express simple and advanced accessibility requirements.

1 Introduction

Access to web content for all is crucial for building the information society. Information on the web should be accessible to all users, independent of disabilities or choice of web browser. Within the EIAO[1] (European Internet Accessibility Observatory) project [2], we want to improve the accessibility of web content by providing measurement data about accessibility. However, it is not straightforward to measure accessibility, because this is very subjective. One main task within the EIAO project is to formalize the informal and subjective requirements.

In order to tackle the problem from a higher level and for making sure the formalization is understood by the experts, a pilot project called MEBACC has been established in cooperation with the Norwegian Directorate of Primary and Secondary Education [1]. The aim of the project is to create a prototype for an Open Source tool for accessibility checking of web documents and web based teaching material. The approach within the MEBACC project is to model the requirements and the measurement policy explicitly. The project is integrated with the ongoing research on metamodelling at Agder University College (SMILE project) and the EIAO project. An important part of MEBACC is to define web document models representing the relevant standards for use in conformance testing. This paper will show how UML can be used for this purpose. A subset of the XHTML specification will be represented as a web document model in UML. XHTML is chosen because it has stricter requirements for structure than HTML; it is therefore easier to create model instances from an XHTML document than from a HTML document.

[1] EIAO has been co-funded since Sept. 2004 by the European Commission under contract number 004526.

The SMILE project is built around a kernel that can represent models on arbitrary levels of the metamodelling hierarchy. We will also show in this article how this SMILE basic representation (called MATER) looks like for the case of UML-like models as used here. In fact, the use of UML is taken here just as one possible representation of the metamodel and the model, because this is a familiar notation. The MATER model abstracts from this kind of representation, such that any notation could be used (e.g. UML).

Within MEBACC, we also built a prototype to show the relevance of our theoretical results. The prototype is simple while still advanced enough to prove the potential of our approach. Carefully chosen subsets of XHTML, OCL and the WCAG 1.0 accessibility guidelines are supported in the prototype.

The article is structured as follows. In chapter 2, we give background information about web accessibility measurements in the scope of the EIAO project. Chapter 3 deals with metamodelling in general and with our SMILE project. In chapter 4 we describe our approach and give a small example to demonstrate how accessibility is modeled using OCL and UML. We conclude the paper in chapter 5.

2 Measuring accessibility

There are defined some standards to measure web accessibility. The WCAG guidelines presented in section 2.1 is one standard that will be used within the EIAO project. EARL reporting (section 2.2) is another standard that will be used to evaluate web pages against the guidelines from WCAG.

2.1 The WCAG guidelines

The WCAG – Web Content Accessibility Guidelines [12] is produced as a part of W3C Web Accessibility Initiative [13], and explains how to make web content accessible to people with disabilities. The WCAG 1.0 includes fourteen guidelines, or general principles of accessible design. The guidelines discuss accessibility issues and provide accessibility design solutions, and they address typical scenarios that may pose problems for users with certain disabilities. Each guideline includes a list of checkpoints which explain how the guideline applies in typical content development scenarios.

2.2 EARL reporting

EARL (the Evaluation And Report Language) [14] is a language to express test results. Test results include bug reports, test suite evaluation and conformance claims. EARL is in a RDF based framework for recording, transferring and processing data about automatic and manual evaluations of resources.

EARL expresses evaluations about all sorts of languages and tools, and could be used to evaluate web pages and web sites against WCAG, and then generate an accessibility report corresponding to the test results.

2.3 The EIAO project

The European project EIAO [2] (European Internet Accessibility Observatory) will assess the accessibility of European web sites and participate in a cluster developing a European Accessibility Methodology. The assessment will be based on the WCAG developed by W3C. The project is carried out in a co-operation among 10 partners in a consortium co-ordinated by Agder University College Norway.

EIAO is carried out within the Web Accessibility Benchmarking (WAB) Cluster together with the projects [10] and BenToWeb [11], co-funded by the European Commission. The cluster consists of 24 partner organisations in Europe.

Among its planned output is a set of Web accessibility metrics, an Internet robot "ROBACC" for automatic collection of data on Web accessibility and deviations from Web accessibility standards, and a data warehouse providing on-line access to measured accessibility data.

EIAO is defining an extensible plug-in architecture in cooperation with W3C and the European WAB Cluster. This architecture will allow exchange of web accessibility assessment modules among different applications. The test modules that are produced based on accessibility models, may implement the EIAO interface, and thereby use the ROBACC crawler of EIAO as a vehicle for testing of a large number of web sites.

3 Metamodelling using SMILE

Our metamodelling approach is done inside the SMILE metamodelling framework; this section describes and introduces some of the basic concepts of the SMILE metamodelling framework.

The SMILE project targets all the levels of the OMG four-layer metamodel architecture; this implies definition of an object representation. FORM [3] was the first definition proposed; it was meant to be used on all the levels of a metamodel architecture and it included the `instanceOf`-relation between elements of to adjacent levels. FORM allowed two levels to be tested for "adjacency" (can one be seen as the model for the other). Its successor MATER (Model All Types and Extent Realization) has been extended with a deep instantiation mechanism; this has been done by supporting definitions of patterns that span multiple levels. Since this paper has another focus instantiation patterns are not described here. MATER is more flexible than FORM allowing different "styles" of metamodelling.

3.1 MATER - Model All Types and Extent Realization

MATER defines a uniform way of representing metadata and object information in a metamodelling environment. This uniform representation is a *level independent representation*, meaning that all levels can be represented with the help of one common mechanism.

MATER is not meant to be a metamodel or a meta-metamodel, it is meant to be "the substance" that is used when a level of the metamodel architecture is made; this

proposed basic representation takes care of what [6] calls *intralevel* instantiation and [7] calls the *physical classification*. The conceptual model of MATER is object-oriented; when instantiated an object graph will be the result.

If a metamodel for relational databases is defined, the model level will define the layout of tables, and the information level will consist of actual tables. In the SMILE metamodelling framework the information level will be an object graph that can be mapped to actual tables, the object graph will have a structure that logically correspond to the tables.

Fig. 1 presents the conceptual model of MATER in UML (how to handle basic types is left out, but [3] demonstrate how this can be done).

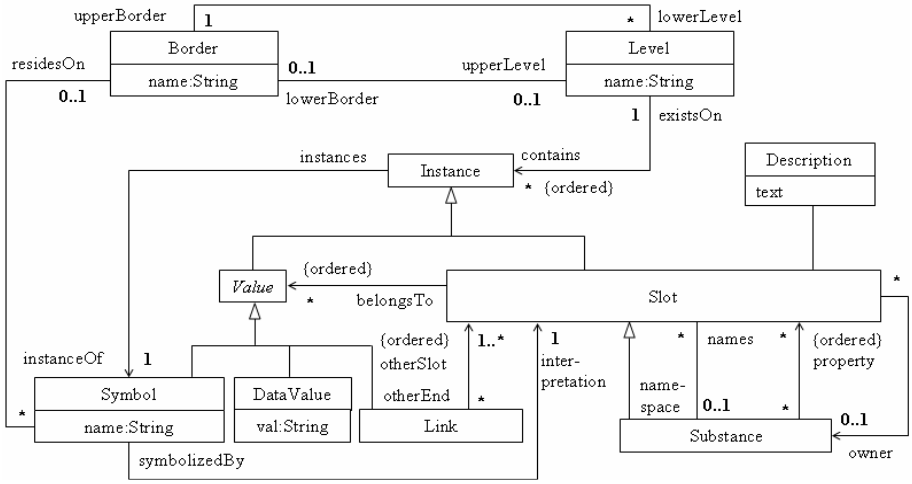


Fig. 1. The conceptual model of MATER.

The metamodel border between two levels is seen as an interface composed of symbols (instances of Symbol) which from the level below represent the instantiable elements of the level above (e.g. names of classes). One metalevel together with the upper and lower interfaces constitutes a manageable module. An instance of `Symbol` that does not reside on the border is *abstract* and will have no instances.

A `Slot`-instance can keep one or more values (e.g. a number); a special type of value is the link-instance which can connect two or more `Slot`/`Substance`-instances. A `Link` instance can represent a reification of an association instance (which in UML is called a link); the Link instance makes it possible to have an `instanceOf`-relation from the "link" to the association which has been instantiated. The “links" of an object graph made by instantiating the conceptual model are called connections; this are not considered objects and must be supported by the underlying software (e.g. references in Java). This problem is discussed in [8] which have the following statements:

... it is possible to reify (i.e. view as objects) links and associations so that they can be modeled as objects and clabjects respectively... The difficulty in reifying links is not in working out how to view them as objects, but in knowing when to stop viewing them

as objects...To break this potentially infinite regression it is necessary to identify certain kinds of links as implicit or primitive links which will not be stored as objects.

`Substance` is a specialization of `Slot`; instances of `Substance` can keep values and have other substance as property; the `owner` and `property` associations used together can define compositions, while the `namespace-names`-association obviously is meant for modelling namespaces. `Description` is meant for additional semantic information.

There is not full agreement on what object-orientation includes and consequently the conceptual model of MATER is one approach; the conceptual model of MATER is kept small but still powerful enough to allow "flexible modelling".

In object-oriented meta-modelling the essential object-oriented concepts must in some way be stated since abstract syntaxes are described with class diagrams. It might be possible to use the underlying representation in such a way that it directly supports a specific object-orientated concept, e.g. that it supports objects with slots. Other concepts might be modelled more indirectly where we as humans must study the structure of several levels to make an adequate interpretation. The following is a list of the "object-oriented" concepts considered and how they can be supported in the MATER approach:

Object: The underlying representation supports this. A class will be described with the help of objects.

Slot: The underlying representation supports this.

Link: The underlying representation supports this.

Multiplicity: Must be modelled explicitly

Identity: A symbol can be used to identify a Slot.

Namespace: `Slot/Substance` has a special association for modelling namespace hierarchies; the members of a namespace instantiate this association to reference the `Substance` which function as a namespace; a member can be a new namespace. A `Slot`-instance that is member of a namespace must have a `Symbol`-instance as value, this value function as a name. The default namespace is the level which means that all symbols that are not part of another namespace must be unique. It is up to the meta-modeller to model the namespace.

Composition: In UML 2.0 class Property has a boolean attribute called `isComposition`; an instance of `Property` typically becomes a property (attribute or "association end") of the owing class (instantiated from `Class`); `isComposite` will be a slot of the `Property`-instance; if the `Property`-instance is an association end and this slot has value true then this is indicated by showing a filled diamond; an object instantiated from such a class will be a container for the object referenced by the slot or value contained in the slot. In UML 2.0 one has to look at the level above to see if something is a composite or not. In MATTER composition can be modelled as done in UML 2.0; additionally the owner-property relations (composite-part) can be used to show the composition where it actually occurs.

Concrete class: Is not directly supported and must be modelled by the meta-modeller. If something is a class then it can be instantiated to objects on the next level - the meta-modeller models this with instantiation patterns, e.g. a metaclass will be

specified on one level; instantiated on the next level to a class; which again can be instantiated to an object on the next level. Examining the levels and how they relate shows what are classes; the names used are irrelevant.

Abstract class: Same as concrete class but the symbol will not be placed on the border but reside "freely on the level".

Property (attribute and association end): Is not directly supported and must be modelled by the meta-modeller. If something is a property then it can be instantiated to slots on the next level.

Association: Is not directly supported and must be modelled by the meta-modeller. If something is an association then it can be instantiated to links on the next level.

Inheritance: Is a description technique; which means that from the "object-level" it looks like ordinary class-instantiation and it is only by examining the model level that the use of inheritance will be revealed. Inheritance is not directly supported.

Packages are a way of grouping elements and defining namespaces. One might consider one level as a package which defines the default namespace. To simplify the presentation packages are not included.

3.2 MATER with set notation.

Fig. 2 shows a concrete notation for MATER; it has similarities with the notation used when visualizing sets, but here extended with meta-information.

Element	Concrete Visual Representation
Border	
Symbol	
Value instance	
Link instance	
Slot (not Substance)	
Substance	
Relations: Instance.instanceOf and Symbol.interpretation	
Relations: Slot.Value, Link.Slot, Slot.owner, Substance. property	
Composition: when Slot s2 is property of Substance s1 and s1 is owner of s2 then s2 is shown inside s1.	
Relation: at the same time Slot.Link and Link.Slot	
Relation: names-namespace	

Fig. 2. A set-like concrete notion for MATER.

Fig. 3 shows how an object of class `Person` and class `Person` can be modeled, corresponding UML notation is also supplied (at level `M2` only UML notation is shown). Class `Person` is on level `M1` and the object on level `M0`. Note how class `Person` is modeled as composite for its property called name, the owner and property associations of basic representation has been used to model this (both being instantiated). Multiplicity information is included for the name property of class `Person` (it is set to 1). Note how namespace information has been modelled by the arrow (the one with the filled arrow head) going from the `Person`-substance to the name-substance (the description of the name property of class `Person`); the value of the `n`-slot (the symbol called `name`) is used as a name in the namespace; class `Person` function as the namespace, in effect all properties of the class has to have unique names.

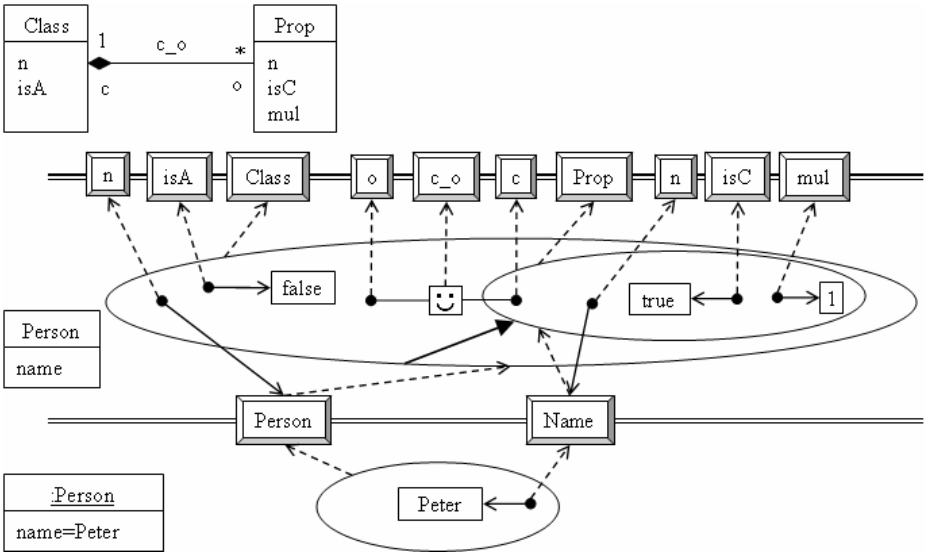


Fig. 3. Two levels with MATER, concrete syntax as shown in Fig. 2

4 Modelling accessibility for XHTML with UML and OCL

The XHTML standard is represented as a UML model called the web document model. The planned tool will take a set of web documents as input and instantiate those into web document model instances based on the web document model. If this instantiation is successful, the web documents are considered valid.

OMG operates traditionally with a four-layer metamodel architecture [5]. For our purpose it is sufficient with three levels shown in Fig. 4. The elements of Fig. 4 are explained below.

Metamodel: The top level is a metamodel that describes the concepts that will be used when defining the XHTML-standard as a model. This metamodel could be the concepts of XML, but we chose an object-oriented approach. We select a subset of the UML metamodel that includes: class, property, association generalization and composition. OCL will work well on such a subset.

Model: The middle level will be an object-oriented representation of a subset of the XHTML-standard itself and can be seen as an instance of the top level. It is at this level the accessibility modeling with OCL is performed; OCL accessibility constraints are attached to the modeling elements and can later be evaluated on the model instance level.

Model Instance: An XHTML-document is transformed to be an instance of the model (level above); the OCL accessibility constraints are evaluated and a report is generated that states to which degree the web document fulfils the accessibility demands.

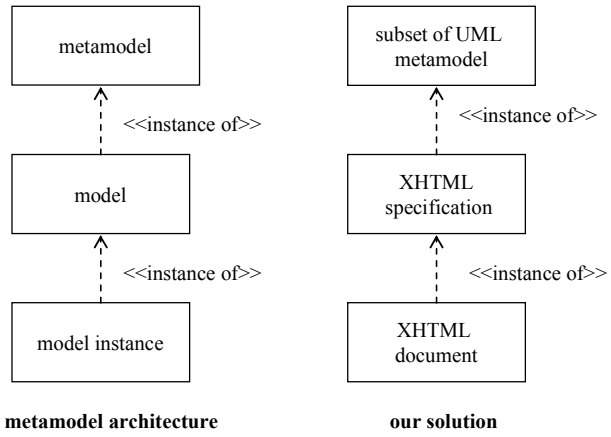


Fig. 4. Our metamodelling architecture

The EARL Evaluation and Report Language will be used for reporting deviations from standards and accessibility requirements. The instantiation technique of MATER will be used when building the metamodelling architecture.

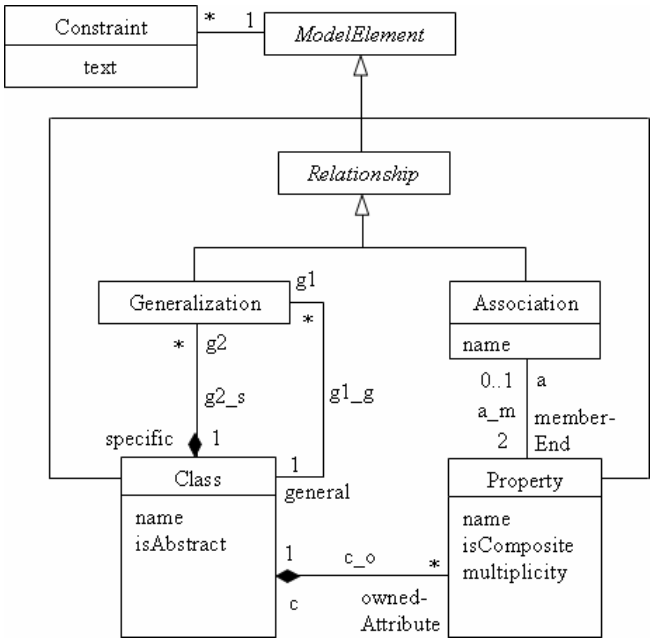


Fig. 5. A minimal reflexive metamodel

4.1 The Metamodel

For our experiments with web accessibility we have developed a very simplistic metamodel as shown in Fig. 5. The metamodel is compatible with UML in that it is just a very restricted MOF (a simplified subset of the UML metamodel kernel), and it is compatible with SMILE as SMILE allows representing it using MATER.

4.2 The Web Document Model (subset of XHTML)

The XHTML standard is represented as a UML model (the middle level) called the web document model. The planned tool will take a set of web documents as input and instantiate those into web document model instances based on the web document model. If this instantiation is successful, the web documents are considered valid.

OCL is a powerful language that offers first order predicate logic on object graphs. OCL expressions can include function-calls with elements of the graph as parameters. The prototype includes a "hard coded" subset of OCL and a set of functions that have been specifically made to do accessibility testing; the functions are available in the evaluation environment and can be used in OCL- expressions.

OCL or OCL-like constraints will be added to the web document model to model accessibility requirements. If there is a valid instance-of relationship between the

model instance and the accessibility model, the tested documents are considered accessible.

We have created a model of a large enough subset of the XHTML 1.0 transitional specification to cover the sample web document. Some more complicated parts of the specification will require OCL constraints, such as the requirement that you must have either HTTP-EQUIV or NAME, but not both, as attributes to a META tag (OCL constraint is not shown in this model).

Accessibility constraints: For the XHTML model given above we formulate three constraints that are derived from the WCAG guidelines.

1. Each image has to have a valid alt tag associated with it. Guideline 1 "Provide equivalent alternatives to auditory and visual content" describes how content developers can make images accessible. Some users may not be able to see images, other may use text-based browsers that do not support images, while others may have turned off support for images. The guidelines do not suggest avoiding images as a way to improve accessibility. Instead, they explain that providing a text equivalent if the image, which serves the same purpose, will make it accessible. An image in HTML has an alt-tag which is used to provide text equivalents.
2. You must have either HTTP-EQUIV or NAME, but not both, as attributes to a META tag. This is in fact not a WCAG requirement but a static constraint for XHTML.
3. The color of the text should have enough contrast with the surrounding color. Guideline 2 "Don't rely on color alone" ensures that text and graphics are understandable when viewed without color. Checkpoint 2.2 says that it is important to ensure that the foreground and background color combinations provide sufficient contrast when viewed by someone having color deficits or when viewed on black and white screen.

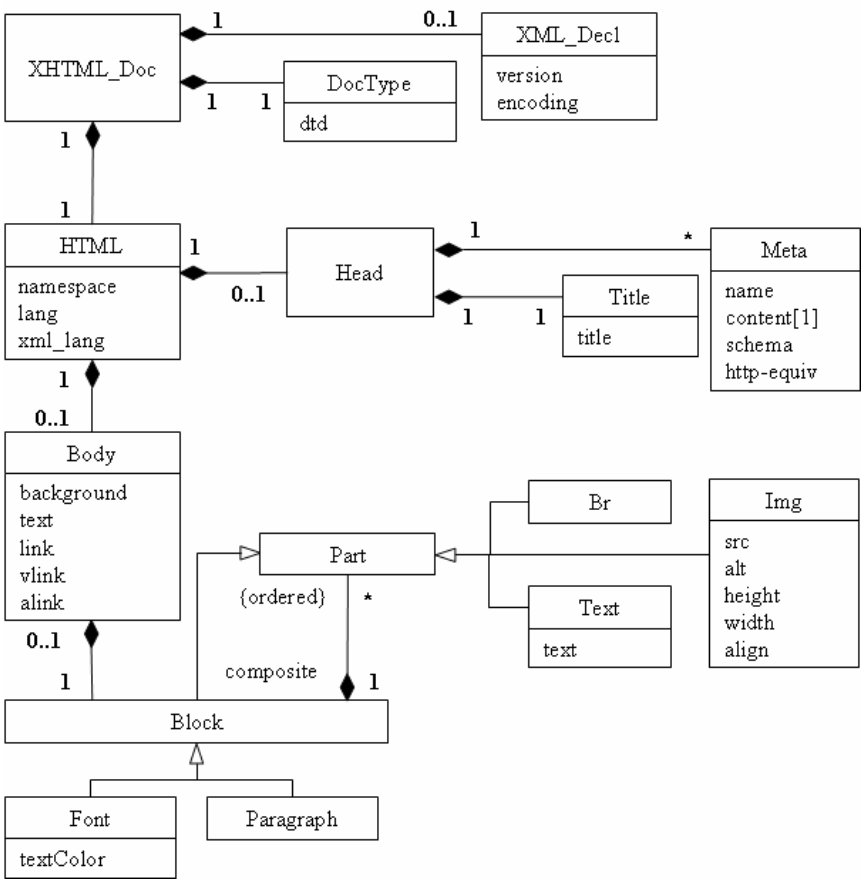


Fig. 6. The web document model

These constraints are formulated in OCL as follows.

1. Context Img
   Inv: libAcceptableAltTag(alt)
2. Context Meta
   Inv: name.size() > 0 xor http-equiv.size() > 0
3. We define an auxiliary recursive function that finds the body.
   Context Block
   def: getBody(b : Block) : Set(Body) =
   if b.body->size() = 1 then body
   else b.getBody(composite)
   endif

   Context Font
   Inv: libAcceptableContrast(textColor, getBody(this)->any(true).background)

The functions libAcceptableAltTag and libAcceptableContrast are library functions.

4.3 The Model Instance

For the presentation we use a simple web page as shown in Fig. 7. We use this web page to evaluate accessibility requirements according to the OCL formulas given.

Fig. 7. A web sample web page

The web document shown above is a (slightly broken) XHTML document, with the following source:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" >
<head>
        <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8" />
        <meta name="generator" content="gedit" />
        <title>Test</title>
</head>
<body bgcolor="yellow">
        <img src="http://www.eiao.net/kickoff_s.jpg" alt="" />
        <br />
        <font color="black">Participants at the EIAO Kickoff meeting in Grimstad,
2004-10-14.</font>
</body>
</html>
```

The document has an empty alt-tag. This is in conflict with the WCAG 1.0 Guideline 1: "Provide equivalent alternatives to auditory and visual content." If the alt-tag was missing, the document would also be in conflict with the XHTML specification.

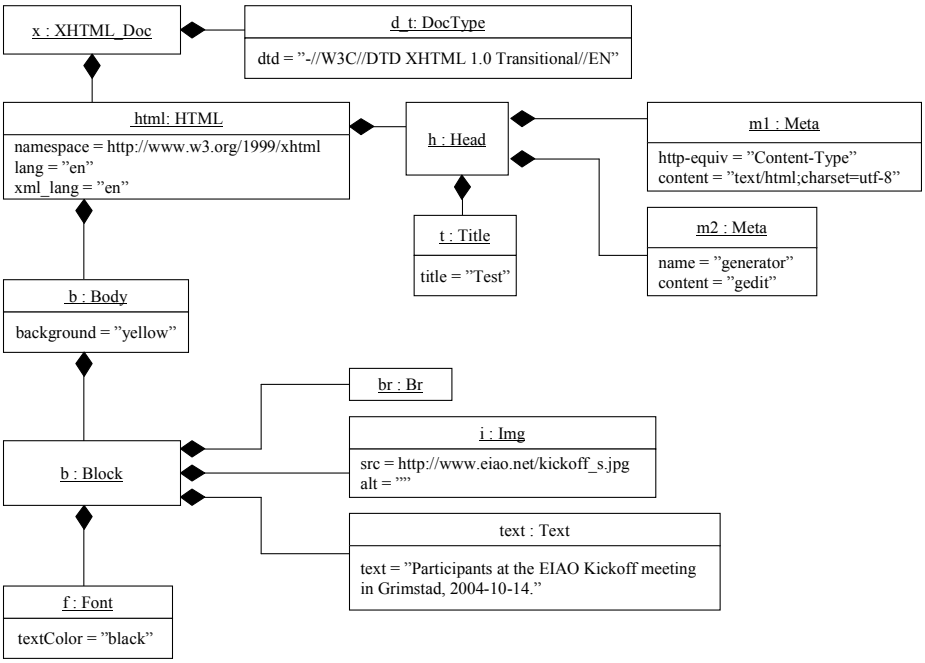
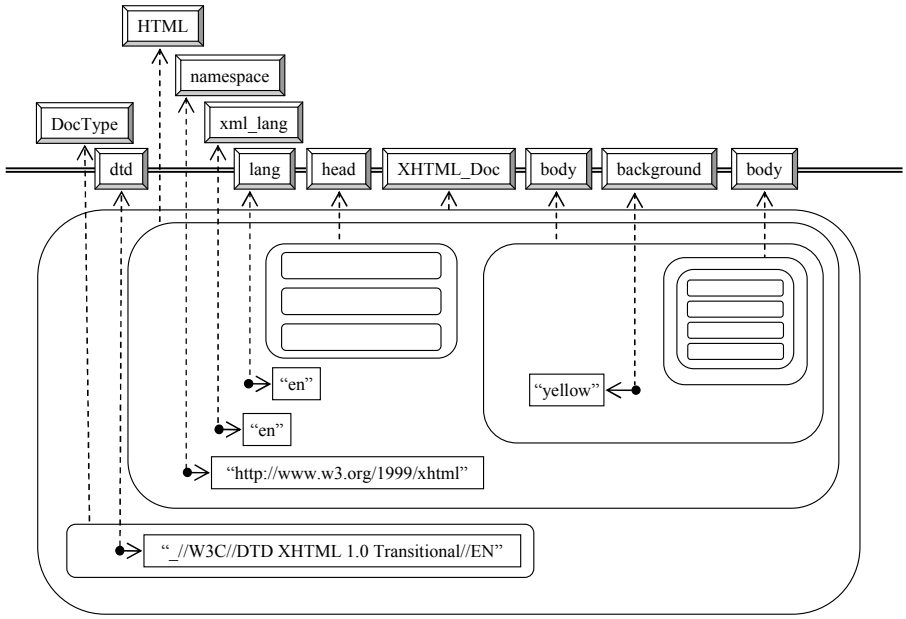


Fig. 8. The sample web page as instance of the web page model

When evaluating the OCL constraints on these instances, we have to check the following conditions.

1. The first condition is applicable in `Img` context. The only instance of this kind is the object `i`. So we have to apply `libAcceptableAltTag("")`, which will give the result `false`. So this requirement is not fulfilled.
2. The second condition is applicable in `Meta` context. We have two meta objects `m1` and `m2`, and for both the condition is fulfilled. So there is no problem here.
3. The third condition refers to the Font context. For the font object f in our example we have to check `libAcceptableContrast("black","yellow")` which yields the result `true`. So this requirement is also fulfilled.

Figure 9 uses the notation of MATER and shows the same as Fig. 8. Figure 9 is not showing all the details: the instanceOf-relation for the links is not shown, and also the details of head and body are not shown.



Figur 9 The instance of the web page model using the notation of MATER

5 Conclusion and further work

Accessibility requirements and web technology are constantly evolving. High level modeling of accessibility requirements can support more rapid generation of new test modules and improve the understanding of the accessibility barriers for web documents. UML seems to be the natural choice for this modeling. However, we need to take into account the limitations of using OCL to define accessibility constraints; we may need to extend OCL to be able to perform some more advanced accessibility tests. The basic representation and instantiation technique from the SMILE project is very useful when it comes to implementation of these ideas.

Starting from the prototype tool we have created, we will extend the subsets of HTML and WCAG 1.0 covered. We will integrate a complete OCL interpreter into the SMILE framework, such that it is possible to express more constraints than just simple comparisons. Moreover, we will extend the library of functions needed to do sensible checks for accessibility

References

1. http://www.utdanningsdirektoratet.no/
2. http://www.eiao.net

3. J. P. Nytun, A. Prinz and A. Kunert: Representation of Levels and Instantiation in a Metamodelling Environment, NWUML 2004
4. http://www.w3.org/TR/xhtml1/
5. www.omg.org
6. R. Geisler, M. Klar and C. Pons: Dimensions and dichotomy in metamodeling, Proceedings of the Third BCS-FACS Northern Formal Methods Workshop, Springer-Verlag1998
7. C. Atkinson and Thomas Kühne: Rearchitecting the UML infrastructure, ACM Transactions on Computer Systems (TOCS), vol. 2, no. 4, 2002.
8. Colin Atkinson. Meta-modeling for distributed object environments. In Enterprise Distributed Object Computing, pages 90 -101. IEEE Computer Society, 1997.
9. Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. IEEE Transactions on software engineering, Vol. 29, No. 5, pages 459-470, May 2003.
10. http://www.support-eam.org/
11. http://bentoweb.org/
12. http://www.w3.org/TR/WCAG10/
13. http://www.w3.org/WAI/
14. http://www.w3.org/TR/EARL10/

GXL and MOF: a Comparison of XML Applications for Information Interchange

Marcus Alanen, Torbjörn Lundkvist and Ivan Porres

TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:{marcus.alanen,torbjorn.lundkvist,ivan.porres}@abo.fi

Abstract. In this paper, we compare the Graph eXchange Language (GXL) and the Meta Object Facility (MOF). GXL and MOF are approaches for information interchange, specifically for the interchange of artifacts created during software development. Although there are several traits in common, some differences can also be found, in particular the more static structure of MOF as compared with the more dynamic nature of GXL. We discuss the benefits and drawbacks of these differences. Additionally we discuss common issues and possible future extensions.
Keywords: Object graphs, Information interchange, MOF, GXL, XMI, XML

1 Introduction

Todays fast-paced technological advancements require a more streamlined way to represent and manipulate information. Our current way of managing e.g. software projects includes several kinds of different information: requirements, specification, timetables, personnel resources, actual source code, test reports, et cetera. All these artifacts relate to each other, but are usually described and manipulated using different data formats and tools.

In this article, we compare two XML [31] applications which have been created to model and interchange data about software and software development artifacts. The Graph eXchange Language (GXL) [36, 35] is a standard exchange format for graphs by Richard C. Holt, Andy Schürr, Susan Elliott Sim, Andreas Winter et al. [11] with the backing of several research communities. GXL is used to describe arbitrary graphs, but additionally it can be used to define GXL schemas which constrain the graphs so that only specific kinds of graphs can be built.

The Meta Object Facility (MOF) [17] from the Object Management Group [15] is a framework for describing metamodels. These metamodel can be used to create models. Metamodels can also be seen as models, with MOF as their metamodel. Serialization is done using the XML Metadata Interchange (XMI) [19, 24] format, which is an XML application. In this paper, we concentrate on the older and significantly simpler MOF version 1.4 instead of the relatively new and complex version 2.0 [20]. The arguments regarding MOF remain mostly the same in any case, although we are aware that version 2.0 includes some interesting enhancements.

As can be seen, both standards employ a way to describe languages (GXL schemas and MOF metamodels) as well as instantiations of these languages (GXL graphs described using the GXL Document Type Definition (DTD) and MOF models described using XMI). We have extensive practical experience in using MOF-based modeling technologies and XMI [1, 2, 5], but lack practical experience of GXL. However, as both standards aim to provide a way to describe interconnected parts of information and thus are quite closely related, we believe we are in a position to compare them.

As we are interested in modeling information, we claim that any standard with sufficient expressiveness for representing information is meta-circular [3, 4], i.e. it should be possible to use the standard to represent itself. In the case of MOF 2.0 [20], MOF 2.0 is used to describe UML 2.0 [21], which in turn is used to describe MOF 2.0, which amounts to the same thing: MOF is used to describe itself. Furthermore, the modeled information conforms to some kind of meta-information model that describes more strictly how pieces of information may be connected. This conformance is represented using meta layers in the modeling community, but similar structure can be found in GXL. Thus MOF is the meta-metamodel, models described using MOF are metamodels (e.g. UML) and finally the creations of the user are models. In GXL, the terminology is different: the GXL metaschema is similar to a meta-metamodel, GXL schemas are similar to metamodels and GXL graphs are similar to models. So, in a way, this article is a comparison of GXL the DTD together with GXL the metaschema and MOF together with XMI.

We proceed as follows. In Section 2 we give an overview of GXL and MOF. We also look at some practical aspects and the current usage of these standards, such as diagram support, transformation technologies, extensibility, et cetera. As both GXL graphs and MOF models metamodels are XML applications, we also discuss their respective serializations. In Section 3 we summarize the presentation by discussing common issues and differences between the two standards. In Section 4 we present some related work and ideas for future work. We finally conclude in Section 5.

2 An Overview of GXL and MOF

For the purposes of this article, we claim that the structure of information can be expressed as graphs. However, these graphs may have constraints on how their nodes and edges can be interconnected.

The benefit is that graph theory has a very strong and well-understood mathematical foundation. In general, a graph consists of two kinds of *elements*: *edges* and *nodes*. These elements are *typed*, *attributed* and *hierarchical*. The type of an element determines its classification. All elements of the same type can be seen as having some structural or semantic commonality. Attributes are key-value pairs of primitive type such as strings which describe the element further. Allowing an element to include other elements (or other graphs) into itself supports hierarchical graphs.

An edge connects n elements together. If $n > 2$ the edge is said to be a *hyperedge*. An edge can also be *directed* which splits the n connections into two nonempty sets, one considered the *source* collection of elements, and the other the *target* collection. Very often edges may only connect to nodes, not to other edges. Edges have additional

properties which describe the *ownership* between the source node and the target node. Edges are often grouped into specific categories depending on which kinds of nodes they interconnect and what the semantics of the edge is. These groups can then be considered *ordered* or *unordered* and the *multiplicity* of the edges is of importance. If several edges between the same source and target node is allowed, the group can be considered a *bag* instead of a *set*.

2.1 GXL

Structure A GXL Node supports directly the properties defined previously for nodes in a graph. GXL supports ownership hierarchies by inclusion of other subgraphs, which contain other nodes and edges. GXL supports binary edges as a special Edge element, and hyperedges with a Relation element. All elements can be attributed via the Attribute element, and all elements support an optional type via the hasType connection to the Type element. The type is defined in a GXL *schema*, which will be discussed later. The GXL graph model arrangement can be seen in Figure 1.

The GXL graph model establishes few restrictions on what graphs can be created and is thereby a very general solution. This can also be seen in its history, where GXL was created by merging properties from several graph formats such as the GRAph eXchange format (GraX) [8], Tuple Attribute Language (TA), and the graph format of the PROGRES graph rewriting system. The only small drawback of such a general solution is that in order to establish more constrained graphs it must be possible to define these constraints in some language. These languages are called *schemas* in GXL. If we also want tools to support generic manipulation of information all of these languages must adhere to some common (meta)language, which is called the GXL *metaschema*. The beauty of GXL is that it describes the schemas and the metaschema as GXL documents. This means that a GXL information processing tool only needs the GXL DTD to load and save GXL graphs, schemas and the metaschema. This arrangement can be seen in Figure 2. Elements in the schemas can be used as types. However, this also means that there is an extra layer of indirection/understanding that tools must perceive, not just the XML document itself. So even though the tool can load arbitrary GXL documents, it must understand the relationship between metaschemas, schemas and vanilla GXL graphs. Failure to accomplish this means that graph modification or querying might not be feasible.

The graph part of the metaschema is depicted in Figure 3. An inheritance hierarchy of the GraphElementClass metaelement with *multiple inheritance* can be created with the GraphElementClass.isA relation. Also some metaelements can be declared *abstract* with the GraphElementClass.isAbstract property. Subgraphs can be created with the hasAsComponentGraph property. These are identified by a name, and have a lower and upper *multiplicity constraint* which tells how many subgraphs of the given name must exist for instances of the metaelement. The order of the subgraphs can also be specified as important with the relatesTo.isOrdered property.

Edges can be of three types: compositions, aggregations and “plain associations”. Also edges have lower and upper multiplicity constraints and can be directed or undirected; both the source and target collection can be ordered or unordered, meaning that order is considered important and must be preserved by any input/output routines and

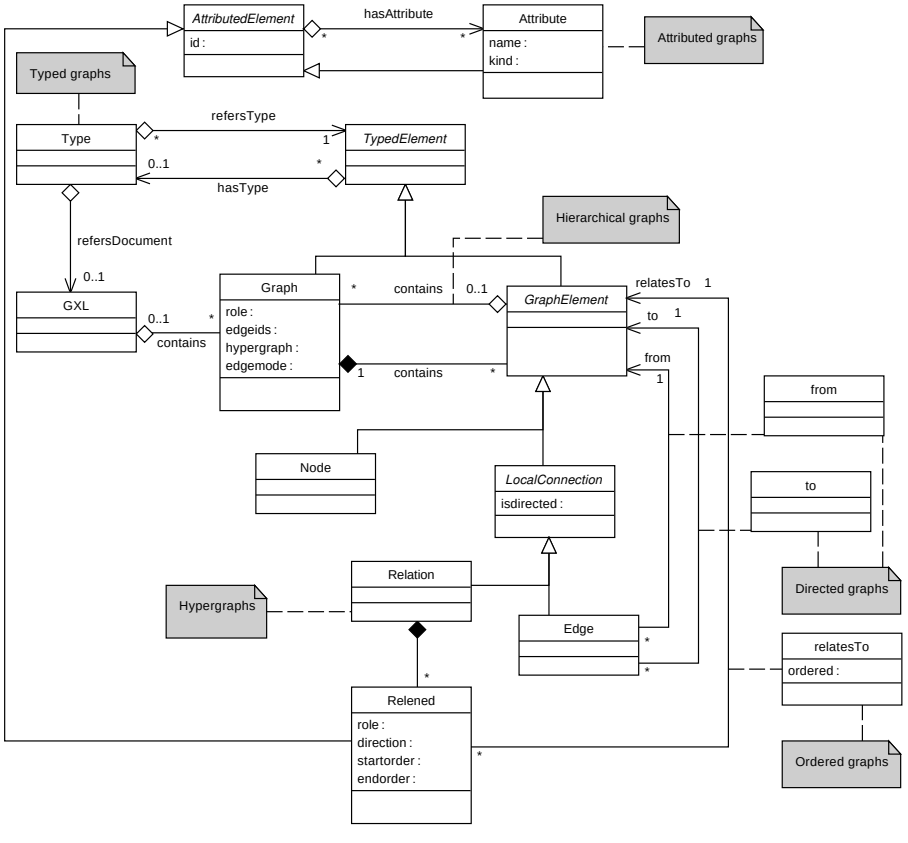


Fig. 1. The GXL Graph Model that defines the GXL DTD. All GXL artifacts correspond to this.

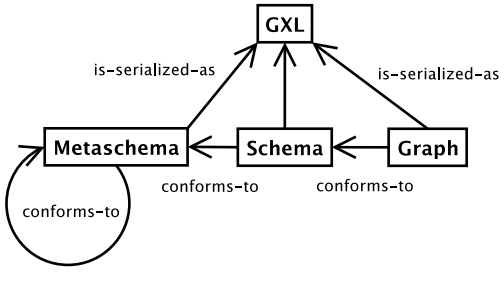


Fig. 2. Overview of GXL and its artifacts. Note how there is only one static serialization format.

must be taken into account by query or transformation algorithms. The edges represent an ownership hierarchy at the graph level, whereas (sub)graph containment represents an ownership hierarchy at the metaschema level and can be used to split schemas into separate “packages” (the subgraphs).

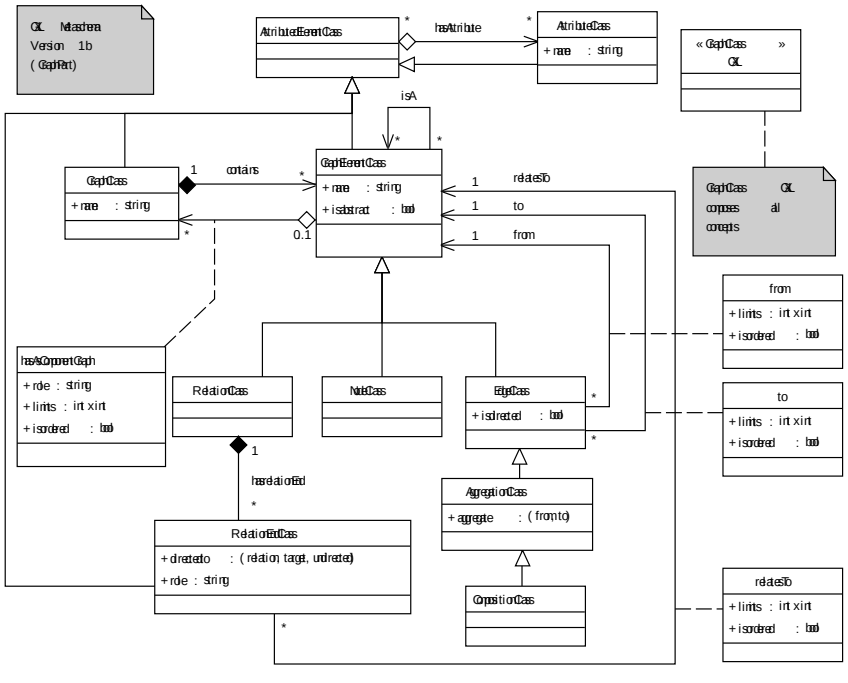


Fig. 3. Part of the GXL Metaschema. Instances of the metaschema define additional restrictions on GXL graphs.

However, the metaschema cannot describe more complicated constraints. This has the benefit that the theory for representing and validating graphs remains fairly simple, although practical considerations might dictate a need for arbitrary constraints. For example, in the definition of the UML metamodel, additional constraints have been heavily used, and thus it does not sound realistic to ignore such a constraint facility.

Element Identification For practical purposes of serialization, elements in a graph may possess an *identity*. In GXL this identity is described with the AttributedElement.id property and is a string unique to the XML document. This is correctly marked as an XML identifier in the GXL DTD, although in the long run the xml:id candidate recommendation [33] might be adopted when or if it is standardized by the World Wide Web Consortium. However, a globally unique name is important because we want to

reference elements from other GXL (or XML) files and a simple local identifier or name is not suitable for that. A more opaque globally unique identifier is necessary.

Schema Identification In order for tools to understand a GXL graph more thoroughly, it is important to be able to identify what schema is being used, i.e. what types are available to GXL elements. The schemas are usually defined in a separate GXL file and shared among all the GXL graphs of that type. Linking to a schema is done using the native facilities of XML, i.e., XLinks [32]. XLink allows the use of Uniform Resource Identifiers (URIs) which can be used to uniquely identify a document e.g. on the WWW. This allows a GXL graph to explicitly reference a specific schema, and additionally it allows tools to download the schema from the location specified by the URI. This means that generic tools can be extended on-the-fly with new schemas.

Visual Representation GXL itself does not define a mechanism for presenting a graph visually on-screen, although this can be remedied in two ways. The simple solution is to define attributes that describe the position, size, form et cetera of a GXL Graph-Element. The more complicated solution is to define a whole new schema for describing the visual representation, thereby decoupling the abstract syntax (the graph) from the concrete syntax (the presentation). This idea is similar to what is already being done by the OMG in the form of the Diagram Interchange (DI) [23] standard and has the benefit that the representation can be split into several possibly different diagrams, each showing a subset of the abstract graph.

Transformation GXL graphs can be transformed with the Graph Transformation eXchange Language (GTXL) [27], although at this moment a revision of GTXL seems to be under way by Leen Lambers [13]. Unfortunately, we do not have experience with GTXL yet and cannot comment on its viability. On the other hand, graph transformations have been extensively researched and we believe it should be possible to adapt any transformation technology using graphs from one underlying schema to another with few problems.

Extensibility GXL allows arbitrary embedding of extra non-GXL information into any GXL node. This has the disadvantage that tools must be ready to process the non-GXL information somehow, either by simply ignoring (and remembering) it or removing it.

Current Support and Licensing Current support of GXL seems to be very good. There are several researchers and companies listed as supporters or contributors on the GXL website [11]. Several tools include export or import capabilities of GXL, such as the round-trip UML software engineering tool Fujaba [14] or the graph transformation toolset GROOVE [26].

Overall, there is activity in the GXL community. GXL is licensed without any fees or restrictions.

2.2 MOF

The Meta Object Facility takes a slightly different approach to modeling than GXL. In MOF, the developer must first define a language (a metamodel) that can be used in creating the actual model (i.e. the actual information). One of the possible metamodels that can be defined in MOF is MOF itself, thereby closing the meta-circularity.

Structure A part of the MOF meta-metamodel can be seen in Figure 4. We have restricted ourselves to the parts that mainly describe the structure of metamodels. As a simple starting point for comparing MOF models to a graph, we may say that the nodes in a graph are mainly Class metaelements, and that edges are represented by Association metaelements.

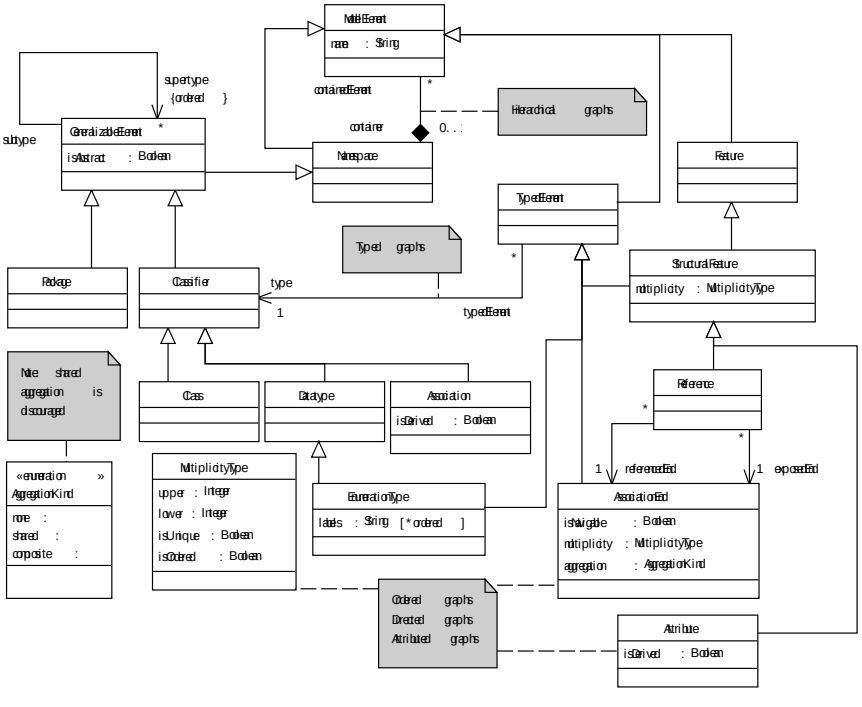


Fig. 4. Part of the MOF meta-metamodel.

It can be understood that a metaelement can establish ownership by two means. One, a metaelement can have Attribute metaelements via the Namespace.containedElement connection. These parts have an obligatory type via TypedElement.type and a MultiplicityType that states the minimum and maximum amount of, as well as possible ordering and uniqueness of, elements. Two, a metamodel can have Association metaelements

which each contain exactly two AssociationEnds. These, almost similar to the Attribute, establish a link between two metaelements, but each AssociationEnd can be explicitly set navigable (which supports directed graphs) and three different kinds of aggregation, along with the usual support from MultiplicityType. However, an Association can be and usually is bidirectional, meaning that if a source element is connected to some target element via their slots, that target element is also connected to the source element.

The three different kinds of aggregation are the same as in GXL: plain, aggregate and composite. However, using aggregation (shared composition) is discouraged and it has been removed in MOF 2.0. The reason might be that if one ignores the plain associations, the resulting ownership structure in the form of composite connections form a tree, which has been found to be a very useful structure and which directly maps to XMI. Aggregation, resulting in an ownership structure of directed acyclic graphs, is not as common, although it can certainly be useful.

So to summarize, an ownership hierarchy of metaelements is established via the Namespace.containedElement property, and as in GXL, it can be used to split metamodels into "packages". An ownership hierarchy of elements is established by Associations with one AssociationEnd marked as composite.

Reference metaelements are owned by Classes and are used to track which AssociationEnds are connected to them. This seems a bit redundant, as the Classes could reference some of the AssociationEnds directly. Thus other more light-weight metametamodel approaches have been created, e.g. the Eclipse EMF [9] or our own Simple Metamodel Description (SMD) language in Coral [1].

Since MOF employs a two-step process whereby the user first creates a metamodel, which then allows them to create models, the resulting usage and serialization of those models (in XMI) is very different from GXL. This is depicted in Figure 5 and shows that tools require metamodel-specific XMI importers/exporters. In other words, to be able to load a UML 1.4 model from an XMI document, the tools must know how to acquire the UML 1.4 metamodel definition first, otherwise it is unable to load it correctly. This is a big contrast with GXL-compliant tools. The GXL tools may be able to load the graph with an unknown schema, although they cannot process it much further.

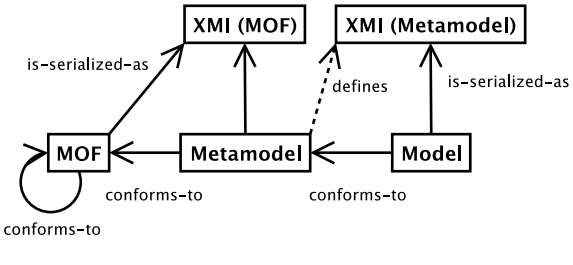


Fig. 5. Overview of MOF and its artifacts. Note how there are several serialization formats on top of XMI. One is the static serialization format, XMI[MOF], and every metamodel defines its own serialization format.

Constraint support in MOF can be assessed as excellent due to the Object Constraint Language (OCL) [16, 22], an addition to MOF. OCL enables a metamodel developer to add arbitrary constraints to the users' models, thus enforcing very sophisticated constraints between elements. A tool can then check these constraints and report nonwell-formedness.

Element Identification Element identification in MOF is handled by XMI with its **xmi.id** and **xmi.uuid** XML attributes. They have been properly defined in XMI and the UUID specification [6] and we have extensively discussed this in [2]. To summarize, elements can be locally identified in an XML document as well as globally with a UUID string, enabling rigid inter-file element identification. On the other hand, current support by XMI exporters/importers is brittle.

Language Identification Similarly to GXL, it is important to detect which metamodel is being used in a model. XMI allows using several metamodels in the same document, and in the new XMI 2.0 standard the XML namespace [30] declaration string describes which language is being used where. This usage is nicely aligned with advances in XML by the World Wide Web Consortium. The only issue is that "there is no requirement or expectation by the XML Namespace specification that the logical URI be resolved or dereferenced during processing of XML documents. [24]" This implies that a tool cannot in general be able to even load a model without knowing the metamodel in advance, because it cannot acquire the metamodel.

Visual Representation MOF does not define a visual representation for models. The basic premise is that there is a strong separation of abstract models containing the semantic data and the diagram which merely display the artifacts on-screen. Thus, the Diagram Interchange (DI) standard [23] has been developed. DI has been successfully used in the Poseidon tool [10] and our Coral tool. This has also been discussed in [2] and the conclusion is that DI is a viable standard that can be used to represent diagram models.

Transformation Even though it is conceivable that several different transformation technologies are used for model transformations, the Query-View-Transform (QVT) [18] is a standard pushed by the OMG to enable the transformation of MOF-based models. As the standard itself is relatively new, we feel it is too early to discuss its benefits or drawbacks.

Other transformation technologies have been described by several authors, for example UMLX [34], YATL [25] and VIATRA [29].

Extensibility MOF metamodels can not as such be extended, but both metaelements and elements can be tagged with arbitrary information using the XMI.Extension XML node. A whole XMI file can be tagged with the XMI.Extensions XML node.

Current Support and Licensing Current support for MOF is low. The meta-metamodel itself has some nonintuitive quirks and is quite big and complex, which presumably has lead e.g the Eclipse team to create EMF. We have also avoided using MOF due to these reasons and opted to explore what fundamental parts are really required in a meta-metamodel. Additionally, MOF 2.0 has become even more complex than it predecessor.

Ironically, the low support for MOF will perhaps not matter, as the serialization is not dependent on MOF per se, but on the metamodels created in MOF. For example, even though our Coral tool is not based on MOF it still is compatible with e.g. the UML 1.4 XMI serialization format.

MOF is released under a royalty-free license.

3 Common Issues and Differences

Comparing Sections 2.1 and 2.2, we can discern several common issues and differences between GXL and MOF. It must be stated that MOF has the backing of an industry consortium which has enabled MOF and related technologies to evolve at a high pace. Examples of these technologies are OCL, DI and QVT, not to mention the flagship metamodel UML, although there is perhaps an ever-increasing fear of a "design-by-committee" syndrome, where a standard reflects only few needs of its users. GXL is more of a community-driven effort where individuals create what they need.

On the metamodel/schema level, both standards have their positive and negative points. MOF has quite a complicated way to describe metaelement interconnections. It even has a second way to establish them, in the form of Attributes, even though an Attribute is basically equivalent to a unidirectional, composite Association. GXL on the other hand does not have inherent support for bidirectional edges (MOF Associations).

GXL contains a crude tagging mechanism in the form of (GXL) Attributes with key-value string pairs (although these do nest). We assume that this concept is included due to the roots of GXL being in describing graphs, which often use attributes for tagging nodes with arbitrary data. Its benefits are not clear for information modeling, especially since a composite edge would mostly serve the same purpose. This is somewhat similar to the Attribute/Association issue in MOF. We feel that perhaps XML itself should employ a standard way to tag elements with extra data.

For modeling information, the choice of having a separate Graph metaelement for nesting is unusual and the benefit is not very clear. A GraphElement could transitively own other GraphElements, without apparent loss of expressivity. This simplifies the GXL graph model and metaschema since it reduces the amount of concepts it must define.

Support for aggregation in MOF has been dropped, which means that there are some information systems that are awkward to describe in MOF. Indeed we have ourselves developed metamodels where aggregation would have been beneficial. Support for aggregation in GXL among multiple files is not without problems, though. For example, it is not clear which file contains the shared subtree of nodes.

GXL has support for hypergraphs whereas the Associations of MOF are restricted to binary edges. We have not found this to be much of an issue when creating metamodels,

but it is worth researching further. N-ary relations are used in the database community for entity-relationship diagrams.

Perhaps the largest differences between MOF and GXL are found in serialization, constraint handling and node interconnections. GXL has only one serialization format, the GXL DTD, which serializes graphs, schemas and metaschemas. This has its advantages, but does require yet one GXL-specific validator for validating the schema-graph relationship. MOF on the other hand defines a serialization format for each metamodel. On one hand there is no extra level of indirection involved, but on the other hand there are multiple serialization formats. So, we do agree with Winter et al that the "XMI/MOF approach requires different types of documents for representing schema and instance graphs" (p. 8 of [36]) and that this indeed is a serious drawback, but only because finding the definition of a previously unknown metamodel is impossible, as has been described in Section 2.2. If the metamodel is known, generic XMI reader and writer routines can be created: e.g. Coral supports reading XMI 1.x and 2.0 as well as writing XMI 1.2 and 2.0 in around 5000 lines of C++ code.

Furthermore, Winter et al claim that "XMI/MOF offers a general, but very verbose format for exchanging UML class diagrams as XML streams (p. 8 of [36])." Our opinion is that the format is verbose for two reasons: bidirectionality, which GXL lacks as such, and named slots, which GXL also lacks. If both of these were added to GXL, it would be just as (or even more) verbose as XMI/MOF. And although XMI requires to use names for all interconnections, we find this to be a great advantage. For example, a class can own a set of attributes and a set of operations in two different slots. The serialization of these elements are cleanly separated into their own XML nodes. Also, navigation via named slots simplifies the manipulation and query of models.

Where MOF (or, perhaps, OMG technologies) really outperforms GXL is in its handling of constraints using OCL. OCL has become well-established in the modeling community and allows additional arbitrary wellformedness constraints to be added to metamodels and models. Naturally, this does not prevent a constraint language to be added to GXL, but the point here is pragmatic: OCL exists currently and is in wide use, whereas we do not know of a similar effort based on GXL.

4 Related and Future Work

Modeling and metamodeling platforms are becoming more of a commodity all the time. A high-level view of the current situation is presented by Harald Kühn and Marion Murzek in [12]. Interoperability between metamodeling platforms is becoming more important all the time. We would thus want to find all the necessary concepts for modeling information. Failure to support a concept directly or by means of a lossless transformation to supported concepts means that transformation of data from (or perhaps even to) that platform is not possible.

Similar views on general-purpose meta-metamodels can be found in e.g. [3] and [28]. In contrast with the meta-circular definition, the work of Thomas Baar avoids the meta-circularity with a set-theoretical framework [4] to describe abstract the syntax of languages.

We plan to research further on this topic, trying to cover other meta-metamodels and other information systems. Examples of such are the Eclipse EMF, the XMF/XCore system from Xactium [7] and our SMD. Our aim is to extract the fundamentals in modeling information from these frameworks. Even though it is not necessary to create a meta-meta-metamodeling language, one emerges as a side-effect from a generic modeling platform, as can be seen in Figure 6. A sufficiently expressive meta-metamodel can be used to create metamodels from the other meta-metamodels, and transformation technology means that all of this ought to be transparent to the end user. The generic meta-metamodel might even be one of the existing meta-metamodels.

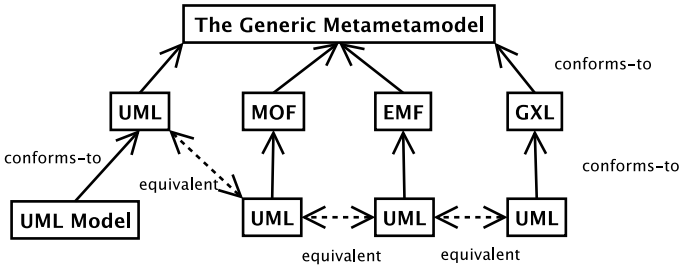


Fig. 6. Overview of how a very expressive meta-metamodel can model all the different meta-metamodels. These can be used to model metamodels, which can be used to create models. The different UML metamodels are equivalent in model expressivity, although the models might be manipulated in different ways since the metamodels are defined by different meta-metamodels.

5 Conclusions

We have presented an overview of two different solutions that can be used to describe information as graphs with nodes interconnected by edges. The Graph eXchange Language has its roots in graph theory and describes every metalevel using the same kind of XML document conforming to the GXL DTD. Additionally graphs must conform to their respective schema which conform to the GXL metaschema, establishing the three meta-layers that is so prevalent in such systems.

The Meta Object Facility has a slightly different approach, by being mainly oriented towards creating metamodels. Using these metamodels, models can be created. Serialization is handled by the XML Metadata Interchange standard. This has the drawback that every metamodel has a different serialization format.

At this point we hesitate to give any judgment on which standard would be more suitable for information interchange. Rather, as can be seen in advances in MOF 2.0, new ways to establish relationships between elements (such as subsets and unions) can and are invented. Therefore there is no perfect solution, but tools and technologies must be able to adapt between different standards. We believe that this is possible by creating

and maintaining a meta-metamodel which encompasses all concepts from the different meta-metamodels and metaschemas that exist today.

References

1. Marcus Alanen and Ivan Porres. Coral: A Metamodel Kernel for Transformation Engines. In D. H. Akerhurst, editor, *Proceedings of the Second European Workshop on Model Driven Architecture (EWMDA)*, number 17, pages 165–170, Canterbury, Kent CT2 7NF, UK, Sep 2004. University of Kent.
2. Marcus Alanen and Ivan Porres. Model Interchange Using OMG Standards. In *Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, September 2005. Special MDE session. To appear.
3. José Álvarez, Andy Evans, and Paul Sammut. MML and the Metamodel Architecture. In Jon Whittle, editor, *WTUML: Workshop on Transformation in UML 2001*, April 2001.
4. Thomas Baar. Metamodels without metacircularities. *L'Objet*, 9(4):95–114, 2003.
5. Ralph Back, Dag Björklund, Johan Lilius, Luka Milovanov, and Ivan Porres. A Workbench to Experiment on New Model Engineering Applications. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2003.
6. CAE Specification. DCE 1.1: Remote Procedure Call, 1997. Available at `http://www.opengroup.org/onlinepubs/9629399/toc.htm`.
7. Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language-Driven Development*. 2005. Available at http://www.xactium.com/.
8. J. Ebert, B. Kullbach, and A. Winter. Grax: Graph exchange format. In *Workshop on Standard Exchange Formats (WoSEF) at (ICSE'00)*, 2000.
9. EMF Development team. Eclipse Modeling Framework. www.eclipse.org/emf.
10. Gentleware. The Poseidon for UML product. `http://www.gentleware.com/`.
11. Graph Exchange Language website. `http://www.gupro.de/GXL/`.
12. Harald Kühn and Marion Murzek. Interoperability Issues in Metamodelling Platforms. In *Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA 2005)*, February 2005.
13. Leen Lambers. A new version of GTXL: An Exchange Format for Graph Transformation Systems. In *Workshop on Graph-Based Tools (GraBaTs) 2004 at Second International Conference on Graph Transformation (ICGT 2004)*, October 2004.
14. Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. Tool demonstration: The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.
15. Object Management Group. `http://www.omg.org/`.
16. OMG. Object Constraint Language Specification, version 1.1, September 1997. Available at `http://www.omg.org/`.
17. OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03, available at `http://www.omg.org/`.
18. OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.
19. OMG. XML Metadata Interchange (XMI) Specification, version 1.2, January 2002. Available at `http://www.omg.org/`.
20. OMG. MOF 2.0 Core Final Adopted Specification, October 2003. Document ptc/03-10-04, available at `http://www.omg.org/`.

21. OMG. UML 2.0 Infrastructure Specification, September 2003. Document ptc/03-09-15, available at http://www.omg.org/.
22. OMG. UML 2.0 OCL Specification, Ocober 2003. OMG document ptc/03-10-14, available at http://www.omg.org/.
23. OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. Available at http://www.omg.org.
24. OMG. XML Metadata Interchange (XMI) Specification, version 2.0, May 2003. Available at http://www.omg.org/.
25. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
26. Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.
27. G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001.
28. Dániel Varró and András Pataricza. Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 18–33, London, UK, 2002. Springer-Verlag.
29. Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.
30. W3C. Namespaces in XML, January 1999. Available at http://www.w3.org/.
31. W3C. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. Available at http://www.w3.org/.
32. W3C. XML Linking Language (XLink) Version 1.0, June 2001. Available at http://www.w3.org/TR/xlink/.
33. W3C. xml:id Version 1.0 W3C Candidate Recommendation 8 February 2005, February 2005. Available at http://www.w3.org/.
34. Edward D. Willink. Umlx: A graphical transformation language for mda. In Arend Rensink, editor, *CTIT Technical Report TR-CTIT-03-27*, pages 13–24, Enschede, The Netherlands, June 2003. University of Twente.
35. Andreas Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.
36. Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

Using UML to Maintain Domain Specific Languages

Mika Karaila[1], Jari Peltonen[2], and Tarja Systä[2]

[1] Energy & Process Automation, Research & Technology Department
Metso Automation Inc.
P.O.Box 237 FI-33101, Tampere,
Finland

[2] Tampere University of Technology
Institute of Software Systems
P.O.Box 553
FI-33101 Tampere
Finland
{mika.karaila@metso.com,{jari.peltonen, tarja.systa}@tut.fi}

Abstract. Domain Specific Languages are typically used to solve relatively focused and specific problems. Maintenance of these specific-purpose languages requires a deep understanding of both the language itself and the domain it is applied for. Designing and maintaining a DSL is typically carried out using specific methods, notations, and tools. Without a common modeling technique and infrastructure, this process is seldom repeatable. Standard notations and commonly used tools can make maintenance process less demanding and help the language designers to rather focus on developing the language than modifying and maintaining the language-specific infrastructure.
In this paper, we discuss the use of UML for maintaining DSLs. The built-in extension mechanism of UML, profiles, is meant for specializing UML to a certain context or domain. Using UML profiles to define DSLs has several benefits. For instance, UML is commonly used and known visual modeling language, and thus makes DSL language structures more comprehensible for non-experts of the domain. Further, various CASE-tools supporting UML are widely available. Using these tools allows the engineers to focus on the development and maintenance of the DSL itself instead of tool development. In this paper we discuss the use of UML to improve a DSL maintenance process. As a practical example, we consider a DSL used at Metso Automation.

1 Introduction

A domain-specific language (DSL) is a language designed to be useful for a specific set of tasks, in contrast to general-purpose languages. Examples of DSLs include spreadsheet macros, YACC [YAC03] for parsing and compilers, Csound [CSO05] language used to create audio files, and GraphViz [GRA05] for defining directed graphs. Furthermore, little languages, macro languages, application languages and very high-level languages can be seen as domain specific languages.

Since the set of tasks to be addressed with DSLs is limited, they do not have to contain all the features that the general-purpose languages have. That is, DSLs can often be simplified and less comprehensive compared to general-purpose languages like C or Java. Furthermore, a typical, well-designed DSL uses the terminology of the problem domain and this way aims to tie together the problem domain and solution domain at the language level. On the one hand, all this makes DSLs expressive in their target domain, as well as easy to learn and understand for the experts of the domain. On the other hand, the language may be hard to use for problems not fitting to the pre-defined domain, and the non-experts may have difficulties in understanding the language and its limitations due to the proprietary constructs and terminology of the language.

As the problem domain evolves, the language designed for that domain must evolve, too. However, the DSL maintenance and development work can be hard and the language designers must be skilled persons. For them, it is not enough to be experienced programmers, but they also have to know the domain well enough to be able to use the correct domain terminology and knowledge when solving the occurring problems. Furthermore, when the language evolves, the applications programmed with the language may have to be transformed from the old versions to the new ones. In many cases the language developers face the fact that there are not enough existing applications to motivate development costs of the needed transformations. The low volume of DSL applications makes it also difficult to have other supporting and analysis tools for the language. The break-even point is actually calculated by the cost of maintenance or transformation work versus the cost of work to re-implement applications.

Unified Modeling Language (UML) [OMG03] has established itself in the software industry as the de facto standard for describing software models. Its success is largely based on the fact that the use of the most relevant diagram types (class diagrams, sequence diagrams, and state chart diagrams) has a long history in software engineering. Another reason for its popularity may be in its multi-purposeness: UML is not a design method, nor is it tied with any specific software development process. It is basically just a collection of often-used notations. This means e.g. that there is a need to define the actual meaning of the diagram and element types used, as well as their possible relations separately for each domain. In UML, this can be done with so-called profiles, i.e. UML profiles provide a way to "customize" UML for a specific domain.

Since UML can be customized to specific domains through the profile mechanism, it can also be used to design DSLs. Having a standard format for that, instead of proprietary grammars, makes the DSL in question also easier to learn and understand for non-experts of the domain. UML is used and supported widely, and it can thus be assumed to be familiar to an average software engineer. The visual notation of UML also increases comprehensibility. Further, the DSL designers and maintainers can benefit from the wide range of tool support available for UML.

In this paper we discuss the applicability of UML for designing and maintaining DSLs. The example DSL considered is Type Definition Language (TDL) that is a core part of a domain-specific visual language, Function Block Language (FBL).

FBL is used in Metso Automation for writing automation control programs. Metso provides automation systems for real-time control of factories and ships all around the world. TDL, in turn, is a meta-language used to define the data structures of FBL.

2. Using UML for designing and maintaining DSLs

2.1 Maintenance Problem in DSLs

The use of domain specific languages is typically supported by a set of tools. Basically, the needed tools can be roughly divided into generation tools like parsers, compilers, interpreters, optimizers, translators, etc., and other tools like validation and analysis tools, editors, etc. While tools exist for these purposes, they are most often language specific. There also are tools that actually combine a set of tools, typically an editor and some generation tools, and form a meta-environment for DSL specification. Examples of these kinds of tools include Meta-Edit [ME05] and Generic Modeling Environment [GME05]. If a tool like this uses also a standard notation for language definition, the language can be (mostly) maintained by anyone understanding the standard notation and tools.

Unified Modeling Language (UML) is a set of standard notations used and supported widely. There are an increasing number of developers that are familiar with UML notations. Furthermore, a wide variety of UML tools are available and the volumes of the users and usage scenarios are big enough to support a great amount of commercial products, open source implementations, as well as tools built for research purposes. The tools include, for instance, editors (Violet [VIO05], Rational Rose [ROS05], Together [TOG05], ArgoUML [ARG05], Poseidon [POS05]), model transformation tools (UMT [UMT05], ATL [ATL05], BOTL [BOT05]), consistency checkers (like Neptune [NEP00], Rational Rose [ROS05], Visual UML [VUM05]), etc. In contrast to DSL tools, which are often proprietary, closed, and platform dependent, a designer using UML can typically use a generic tool, choose the platform she wishes to use, and in the case of open source tools even look inside the implementation if she wants to.

A domain specific language must evolve according to the domain it is designed for. A maintenance task may concern the DSL itself, the tools used, or both; any kind of modifications to the language or the tools may be required, including even changes in the language grammar and semantics. If the tools and specification mechanisms are proprietary, it may be very difficult for a new designer to maintain the tools or language, since she probably does not know the notations, the code implementing the tools, etc. Thus, standard notations and tools would help in the maintenance work. The use of standard notation could even diminish the use of proprietary tools and help designers to concentrate on the language issues instead of tool issues.

2.2 Metamodels

Basically, any programming language could be specified by defining the lexical elements, as well as the static and dynamic syntax and semantics. Whenever these definitions are given, another language is used to specify them. This language can be seen as a meta-language from the point of the specified language. These kinds of meta-languages include, for instance, regular expressions, state automata, and class diagrams. Even though e.g. run-time semantics is an interesting subject in DSL definition, we concentrate purely on the grammatical issues in the rest of this paper.

Metamodel is a model of another, lower level model. Accordingly, the vocabulary of a metamodel consists of the concepts (or types) used to describe the lower level model. Therefore, a metamodel can be used to form a language that can be used for modeling. UML is an example of a set of languages defined by their metamodel. The metamodel consists of the abstract syntax, additional constraints (called well-formedness rules) and semantics of the constructs of the UML. Abstract syntax is depicted as a set of class diagrams that present the elements of the UML and their relations. The elements describe the fundamental modeling concepts of UML, and thus form the vocabulary of the UML. The relations between the elements and the well-formedness rules restrict the use of the vocabulary. An excerpt of the UML metamodel specification is given in Figure 1. As an example of the constraints implied by the metamodel, an association must have at least two association ends connected to exactly one classifier each. Each association end may also have navigability, visibility, multiplicity and ordering.

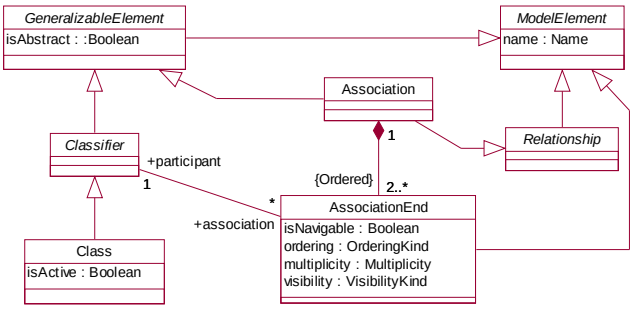


Figure 1 An excerpt from the UML metamodel

Similarly, as a metamodel defines the concepts used in a model, metametamodel defines concepts used in a metamodel, i.e. metametamodel form a language used for metamodeling. The different (meta)levels of modeling form a metamodel architecture, where each level is defined by an upper (meta)level. There are typically four levels in a metamodel architecture: *instance level*, where there are the actual instantiated model elements (e.g. run-time language constructs), *model-level*, where there is the specification of the actual model (or program), *metamodel-level*, that defines the language used for modeling, and *metametamodel-level*, that defines the language used for language definition. Metameta level language typically defines itself. Meta

Object Facility (MOF) [MOF] is a metameta level modeling language used in, for instance, specification of UML metamodel. MOF uses UML class diagrams for specifying the modeling concepts and their relationships.

2.3 UML in DSL Definition

UML is basically just a collection of often-used notations. This means, e.g. that there is a need to define the actual meaning of the used diagram types, as well as the used element types and their possible relations in them, separately for each domain. Basically, this can be done by using MOF to create a metamodel that is either a completely new one, or a specialization of the UML metamodel. Another specialization mechanism, built in the UML itself, is using so-called UML profiles.

In general, metamodeling approach is a good and easily understandable way to specify grammars for languages. For instance, use of MOF is no way restricted to specification of similar languages found from UML, but grammar of any language could be specified with it. The problem of modifying the existing UML metamodel, or specifying a new metamodel with for a DSL specification is that the extensive tool support built on UML does not apply to all languages specified with MOF. Hence, if MOF is used in definition of a DSL, the situation is currently only marginally better than using the proprietary notations and tools.

UML profiles provide a more UML-centric way of defining specializations of UML notations. A profile consists of a set of stereotypes that specialize existing UML metamodel elements by giving a new meaning for them. In addition to the new name and description, a stereotype can define tags and constraints for the elements. In principle, a profile can be seen as a mapping from a metamodel to the UML metamodel. However, a profile cannot restrict or remove existing elements, nor present completely new element or diagram types in UML. The only thing that can be altered is the addition of new meta-elements defined using existing meta-elements. On the one hand, this limits the possibilities of specifying the DSLs, but on the other hand, it makes it possible to use the existing tool support in any of the specified profiles.

In addition to other limitations, UML, and therefore also any UML profile, is somewhat restricted by its object-oriented paradigm. There are extensive amount of languages that do not fit to that paradigm, and thus there is really no point in trying to fit them in to a UML profile. In addition, even though, there are CASE-tools that support the definition and validation of profiles, not all the current commercial tools have these kinds of facilities. The situation is slowly changing, but currently separate tools must be used, for instance, for the validation of the models against profiles. Furthermore, there is a need for transforming the models from the UML tools to, e.g. the compiler of the DSL.

Despite all of the restrictions, UML profiles are a valid approach for defining languages, which paradigm matches with the existing notations in UML. These kinds of languages can, not only be defined, but also used, specialized and maintained using the existing UML tools. The existing tool base also provides new possibilities, for

example, for analysis, visualization, and documentation of the programs built with a DSL.

3 TDL – A sample DSL

3.1 An overview of FBL and TDL languages

Function Block Language (FBL) is a domain-specific visual language, used in Metso Automation for writing real-time automation control programs. The history of FBL goes back to 1985. In that time it was reasonable to build an own limited DSL language with own semantics for such purposes. One of the main motivations was that the end-user was not required traditional programming skills. Therefore, FBL was designed to be a simple and easy to use visual language. FBL, as its name indicates, heavily relies on programming with so-called *function blocks* that are subroutines running specific functions to control a process. The symbols used in a function block diagram are defined by type definitions, specified with another domain-specific language, Type Definition Language (TDL). In our small case study on using UML to design a DSL we use TDL.

With FBL, engineers can design visual programs that connect physical electrical measurement signals to program parameters. Those parameters are referred by symbols. By connecting these symbols the engineer can create algorithms to control and run actuators, such as valves, motors, and pumps, in the process. A simple example in Figure 2 contains an input symbol to read a water level measurement in a water tank. That input symbol could be connected to a *function block symbol* representing a subroutine for calculating and keeping the level. Then the function block symbol is

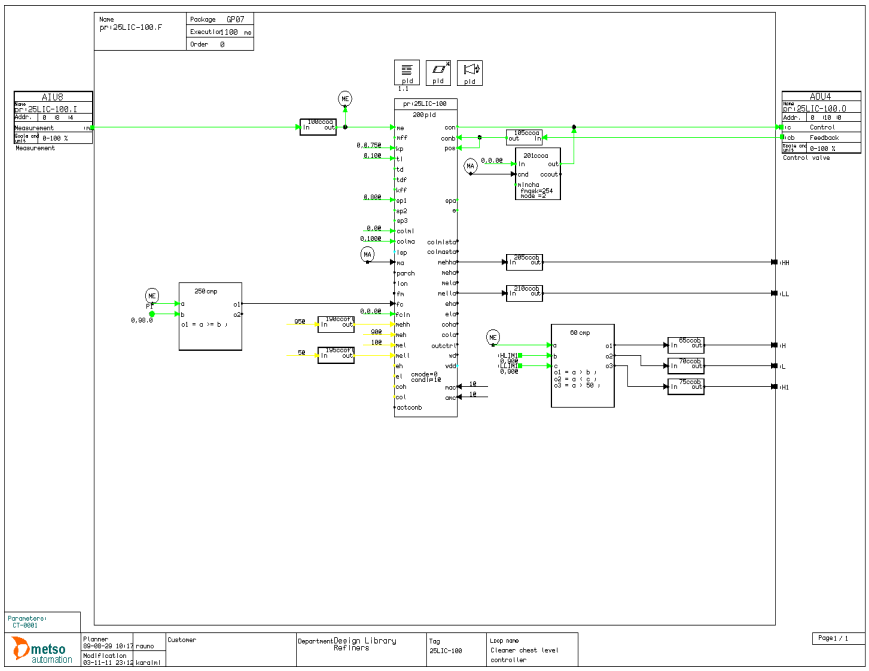


Figure 2 A low-level visual program shown as a Function Block Diagram. The program is for controlling a level in a tank.

default values), and at run-time to control accessing of member values. This metadata is related to the domain TDL is targeted for, namely, automation control systems. For instance, if a "role" of the type is a *function,* then the type is encapsulating a function block that runs some actions according to the data values. Thus, a type not only contains data, but can also be used for controlling the data flow in the real-time environment. In addition to the domain-related metadata, TDL types may also include metadata concerning protection of reading and writing of data.

TDL is a compact language, containing only about 50 tokens (reserved words). TDL is also easy for a programmer to read and understand. Current TDL environment is still almost as it was in year 1985. The designer has a separate editor that can check syntax and semantics of TDL definitions. The editor is not anymore used in interactive mode, but it can be used in batch mode to compile all type definitions. The end user (application engineer) who builds FBL programs does not define or use TDL. All the tool programs use the type definitions that are stored in a database and in binary files. Binary files are used in the real-time environment with optimized set of types. With the reduced set of types system does not need a lot of memory.

3.2 Maintenance requirements and problems

The current maintenance process of TDL types is shown in Figure 3. When a new type is defined, a specification and a requirement document are first written. Then, the TDL definition is made and the implementation of the type is coded. Finally, the new definition and the actual implementation can be tested and validated. Due to the fact that the current TDL environment, i.e. the tools and notations used, are proprietary, there are numerous problems in this process. Even relatively small maintenance tasks, like adding new basic types, are difficult using the current TDL environment. The problems in, and requirements for, maintaining TDL are discussed in what follows.

Type designer must learn the syntax of TDL and be familiar with the principles of using TDL. For building a new type the designer usually needs other, already existing types. Thus, it must be easy to load other types needed to check that the new type is defined correctly.

The type must be defined before it can be used as a part of another type. In TDL a certain parameter value will effect to the following parameters. If a type has a role "data", then all the members also have to be defined to use role "data". It is rather

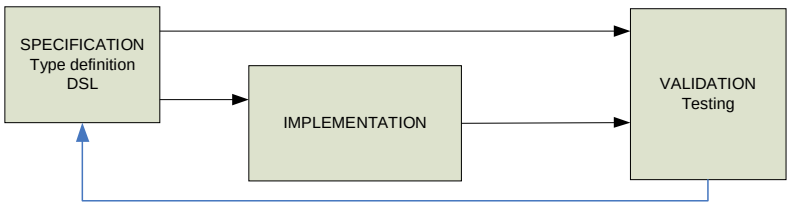


Figure 3 The current development process for TDL types

problematic to define the semantics for TDL and to understand how dependencies inside TDL definitions are build. This originates from the fact that all the dependencies are named: the names are used to refer to the corresponding structures, which hides the actual structures and thus makes TDL definitions less comprehensible.

Usual modifications needed to maintain TDL include adding parameters and modifying the implementation to run more functionality by a new parameter value. The most typical maintenance task is to implement a new type from an existing one. Naming the new types should be done with care, because the user typically associates the name of the type with its usage. Also, naming conventions used should be followed. For instance, new versions of the type are typically named by appending a version number to the name of the extended type. At the implementation level we have to do the corresponding modifications.

Because TDL defines mission critical data structures, a transformation from the definition format to binary format should not produce any errors. To ensure a 100% valid transformation we have to automate the transformation and have a mechanism for running detailed comparisons.

A procedure has been implemented to write all the metadata of existing type definitions into a file in a simple format. It has been used to compare all release modifications. It allows us to transform types from an existing language to a new format and reload all these new type definitions to a database and a binary file. Both of the formats are needed, since database types are used in the engineering phase and binary files are used in real-time. After that we have to dump definitions out again and compare results to make sure we can load all the types correctly in the new format. This is rather complicated and not very intuitive. A more descriptive way to compare different versions of type definitions would be desirable.

We train new programmers to learn and work with FBL and TDL. Training helps the users to learn the language and its semantics more quickly. Such small changes as adding a member or a metadata should be made as easy and fast as possible in the environment used for developing and maintaining the languages. Current problems originate from the fact that FBL and TDL have been used over 20 years and the tools are also as old. It is hard to modify languages themselves because they have been used such a long time; there are more than 2000 implemented and still existing components in TDL. These TDL components, types, are then reused in FBL level by factor over 1000 in each customer project. There are several projects every year and there have been FBL based project since year 1988. The amount of type instances is huge. Especially due to the long history of the languages, their easy maintenance and evolution is essential.

4 UML-based maintenance of TDL

We next present how the current version of TDL grammar can be modeled as a UML metamodel and how the constructed metamodel supports re-design, as well as other maintenance tasks of the language. The metamodel is depicted in Figure 4.

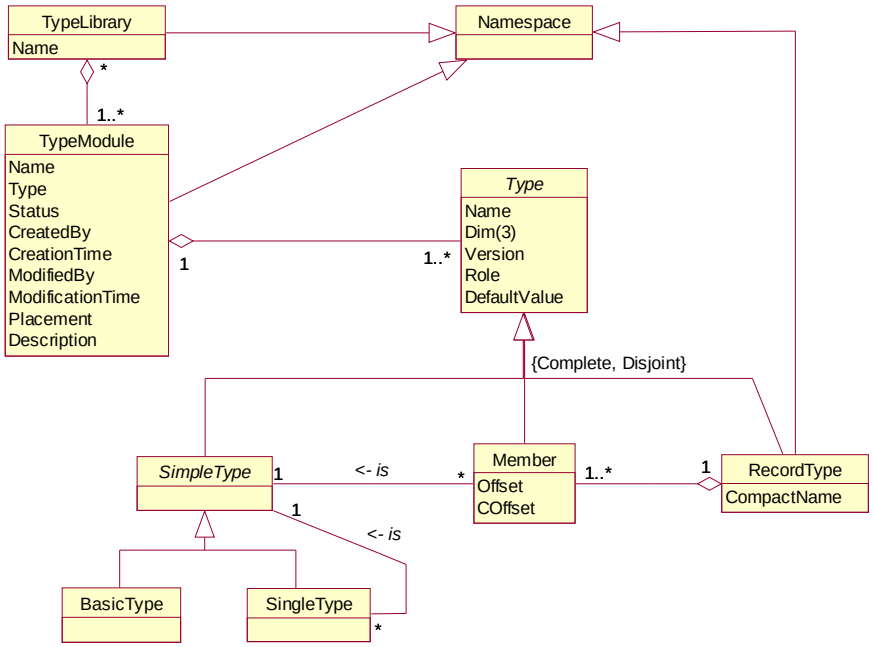


Figure 4 Type definition metamodel

There is a need to make a smooth transition from the old TDL maintenance environment to the new one. The process itself must support good documentation and validation to keep TDL and FBL quality level high. Different kinds of transformations are needed to connect the "UML world" with the TDL usage environment. The first task is to construct a UML metamodel for the language, TDL in our case. The second task is to implement transformation from the textual format (original TDL) to XMI. This is the critical part, requires manual work, and must be tested thoroughly to ensure the quality of type instances. No data loss is acceptable.

In the new maintenance process, shown in Figure 6, we use UML metamodel as the central point to integrate the specification construction and language validation activities. Transformation tools are needed to make a transformation from the constructed metamodel to a TDL definition. This can be done e.g. by storing the metamodel in XMI format and then use XSLT transformations. The TDL definition itself can change later on, as well as the type instances. These changes are marked in the Figure 6 as "updates". The UML environment should support both language structure and instance changes. Changes in the language, either defined by modifying the metamodel or the TDL definition itself, should be automatically propagated to TDL instances (arc "upgrades" in Figure 6), because no bugs in mission critical controls can be allowed and because updating thousands of instances manually would be error-prone and inefficient. Automatic upgrades are indeed possible, since defining default values for all new types is mandatory. In the instance level, changes can and often are made as well. When updates are made, e.g. type members are changed, an object model is constructed and can be then validated at UML-level to ensure that the

new values are legal according to the metamodel. The new design process relying on using UML is shown in Figure 5.

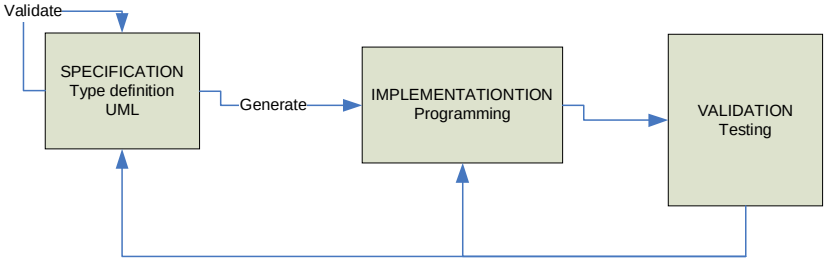


Figure 5 New design process with UML

Because we create new types and edit those definitions either in textual format or UML format there is a need for transformations from TDL definition files to UML presentation. This transformation should not need any extra coding or major maintenance work in the future. One possible solution for a smooth translation is to use an existing parser to read definition files and produce XML as output. One benefit of relying on XML is that we do not have to write a new parser every time we modify the syntax. This will solve the problem to have extendable language. Also, standard tools, such as XSLT, exist to support transformations. After each extension in the language, work is needed to modify all existing instances to be valid. For validation, we can use UML tools and generators to study instance diagrams and to test a new specification before implementing the type.

TDL metamodel is shown in the Figure 4. The metamodel itself is very simple. Types are defined in own type modules that contain administrative attributes. All the modules are included in a type library. The module and library thus define packages for types (kind of namespaces). The types in TDL are actually built from the basic types. There can be only three kinds of instances from the abstract class type. A single type is always constructed using another type as the basis of the type. A record type is a composition of other, so-called member-types. The member types are also

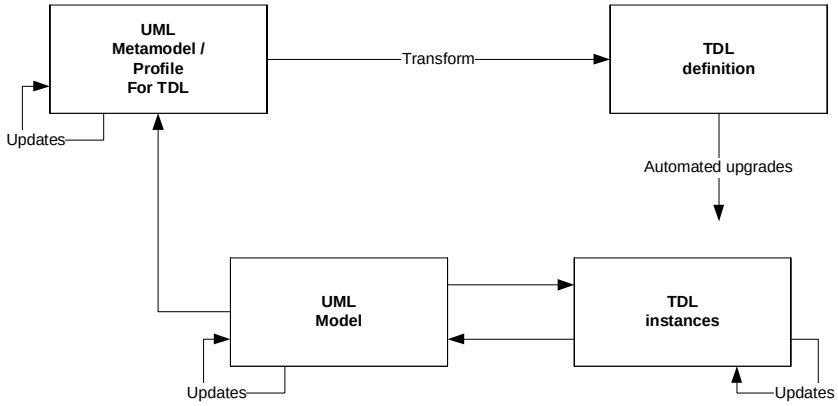


Figure 6 The maintenance process in UML environment

based on other types and must be defined before using them. Members are for defining parameters for FBL function block symbols. These parameters are used inside a function block implementation to control run-time features like calculation or alarming.

The benefits of using UML include having a standard notation and tools available. First, UML CASE-tools can be used to design and maintain the UML representation of the metamodel. Second, UML provides a graphical notation that is easier to any software engineer to comprehend than the proprietary textual notation. Third, validation of the language can be carried out at UML-level by first transforming a TDL definition to a UML model that can then be validated against the meta-model. Fourth, UML is more familiar to new programmers than TDL. It is also easier to use UML for specifying TDL than the proprietary textual format. Furthermore, it can be used to learn the context and principles of how new types can and should be constructed.

While relying on the standard technologies such as UML and XML have clear benefits, also some problems arise, e.g. adding semantics to UML and XML. The old parser currently used validates both syntax and semantics. The language grammar is implemented with a specific tool that contains syntax map with semantic actions. The parser reads type definitions according the syntax, and after a lexical tokenizer is executed, parser called subroutines for semantic actions to check and validate additional semantic rules. Those subroutines collect and save information to a stack to validate the semantics.

UML is planned to be used for designing the language metamodel that allows extending it in the future conveniently. UML was selected, because it is the current de facto standard for modeling and designing software systems. XMI, in turn, was selected because it is the standard XML-based exchange format for MOF-based metamodels, such as UML. Also, XML is well known and widely supported. In addition, there is a wide spectrum of XML processing tools available that e.g. unburden us from planning for maintenance support for the parser. Using these technologies allows us to concentrate on the data model instead of the language syntax. The new process also makes it more possible to reverse-engineer current definitions and to re-engineer new ones. The use of UML also allows us to benefit from various UML model operations developed. For instance, two versions of the TDL grammars can be compared by applying UML set operations [SEL03] on the corresponding UML metamodels.

The new process makes the evolution of the target domain faster. The transformation from the current TDL is written only for supporting the required interfaces: database and binary file. The database and binary files are used by other tools and programs in the framework. The other interfaces were not detected and there were no code body generation tools in 1985. The UML modeling framework opens new possibilities to study, discuss and test extensions without any heavy development work. The language extension can be modeled and people have (more probably) the same understanding when UML based notation is used.

UML tools that could fit to our purposes are currently available. As an example, UMT [UMT] tool is able to read a metamodel in XMI format and use defined trans-

formations to generate target format files (like XML or binary). The support for such transformations is essential, because otherwise we will have the same problem that we currently have but in another format. The current textual format is not supported by any commercial parser. As FBL is a visual language, so should TDL. This would help people to define structures and understand corresponding references that are currently available only as names. Another important issue is that the tool should be extended and integrated to the development process. Otherwise development people will not use the new tools. Thus, from our point of view, a tool should be customizable to fit to the above-mentioned purposes of ours.

5 Related work

Designing of a DSL is typically carried out using specific methods, notations, and tools. Without a common modeling technique and infrastructure, this process is seldom repeatable or reusable for designing other DSLs. The history of DSLs is long. In fact, before identifying common programming concepts and abstracting them as general-purpose languages, early computer languages were indeed application-specific. Despite a long history of DSLs, their systematic study is quite recent [NEI80].

While general-purpose programming languages such as Java or C++ were designed to be appropriate for virtually any kinds of applications, DSLs simplify the development of applications in the specialized domain [CE00,CZA05]. This often comes with the cost of generality: DSL programs can be composed partly in a generative manner, but the applications constructed using the DSL in question are of specific purpose and belong to a specific domain. In fact, according to Czarneck [CZA05], DSLs belong to the set of basic concepts and ideas of generative software development, together with domain and application engineering, generative domain models, networks of domains, and technology projections.

Domain Specific Modeling (DSM) aims at creation and use of (typically graphical) DSLs with domain-specific generators that create full production code directly from models [KT00, PK02, GEA04]. In some respect, DSM is thus comparable to use of visual DSLs. DSM have been recently successfully applied e.g. in the domain of embedded systems, to faster develop variants belonging to the same family of systems. [KEA96, LEA98, SZI98]. Some tool support for DSM is currently available. MetaCase tools, such as MetaEdit+ [ME05] and ObjectMarker [MV02], for instance, provide means to support DSM and create and use DSLs. MetaEdit+ offers a metamodeling tool suite for defining modeling concepts, their properties, associated rules and symbols, needed to specify and implement modeling languages. GME (Generic Modeling Environment) is another environment, closely related to MetaCase tools, for creating domain-specific modeling and program synthesis environments [GME05].

Since executable programs are directly generated from visual FBL programs that are also called FBL diagrams, FBL programming is comparable with DSM. In this paper we discussed UML support for maintaining DSLs. We demonstrated this by

introducing a UML profile for a sample DSL, namely TDL, which is used to define type definitions used in FBL symbols. Like GME, the proposed approach relies on UML-based metamodeling. The use of UML for defining DSL grammars not only provides an intuitive graphical representation of the language grammar, but also allows the DSL developer to benefit from UML model operations. For instance, two versions of the DSL grammar can be compared by applying UML set operations [SEL03] on the corresponding UML models, and the differences can be easily identified from the resulting model.

In [SPI01], Spinellis proposes eight design patterns, reflecting eight common ways to design and implement DSLs. These patterns can also be used e.g. for managing the evolution of DSLs. Spinellis proposes e.g. a *language extension* creational pattern for adding new features to an existing language, a *piggyback* pattern for using an existing language to implement DLSs, and a *language specialization* pattern for removing features from a base language to form a DSL. Spinellis rather focuses on the DSL development process than proposes a modeling technique or infrastructure for actual DSL implementation. In this paper, we propose using UML metamodeling to support designing of DSLs. Further, tool support can be provided, relying on general purpose UML model processing techniques, to support maintenance of the DSLs. For instance, changes in the language e.g. due to adding new features to it, can be identified and visualized by applying UML set operations [SEL03] to the UML representations of the DSL grammars.

6. Concluding remarks

DSLs are typically designed and used for specific set of tasks and the languages are applicable in rather limited domains. This on one hand enables their effective use for generative software development, but on the other makes the language difficult to understand and learn for engineers not knowledgeable of the domain. Also, the tools used are often specially built to only support the languages in question.

A need to have more support for DSL modeling has be acknowledged. Also, transforming from proprietary solutions to more general ones is aimed for. Maintenance and redesign needs and activities are rather common, since the domain languages evolve according to the domain itself. This requires the DSLs to be easily maintainable and extendable. In practice, however, this is not always the case. The maintenance work could be decreased by using standard solutions and components that are commonly available.

In this paper we discuss the role of UML to support DSL design and maintenance. The benefits of using UML instead of e.g. proprietary grammars include increasing the comprehensibility of the DSL in question, esp. for non-experts of the domain, and allowing the language designers and maintainers to benefit from the a wide range of UML tools currently available.

As a practical example, we consider the use of UML for designing a DSL in a small case study. Function Block Language (FBL) is a domain-specific visual language, used in Metso Automation for writing real-time automation control programs.

The symbols used in FBL programs are defined by another domain-specific language, Type Definition Language (TDL). In our small case study we use TDL as a sample DSL. We first construct a UML metamodel for TDL. We further discuss the possible maintenance scenarios that can be supported by the use of the constructed UML metamodel.

References

[ARG05] ArgoUML, http://argouml.tigris.org/, 2005.

[ATL05] The ATL home page, http://www.sciences.univ-nantes.fr/lina/atl/, Inria, 2005.

[BOT05] The Botl tool, http://www4.in.tum.de/~marschal/botl/index.htm, 2005.

[CE00] K Czarnecki and U. Eisenecker, *Generative Programming,* Addison-Wesley, 2000.

[CSO05] cSound, http://www.csounds.com/, 2005.

[CZA05] K Czarnecki, Overview of Generative Software Development, 2005, on-line article, http://www.swen.uwaterloo.ca/~kczarnec/gsdoverview.pdf.

[GEA04] J. Greenfield, S. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley, 2004.

[GME05] GME: The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/, Institute For Software Integrated Systems, 2005.

[GRA05] Graphviz – Graphical Visualization Software, http://www.graphviz.org/, AT&T Research Labs, 2005.

[KT00] S. Kelly and J-P Tolvanen, Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, J. Bezivin and J. Ernst eds., *Proc. of International workshop on Model Engineering, ECOOP 2000*, 2000.

[KEA96] R. Kieburtz. et al., A Software Engineering Experiment in Software Component Generation, *Proc. of 18th International Conference on Software Engineering*, IEEE Computer Society Press, March, 1996.

[LEA98] E. Long, A. Misra, and J. Sztipanovits, Increasing Productivity at Saturn, *IEEE Computer*, August 1998, pp. 35-43.

[ME05] MetaCase Consulting, MetaCase Consulting website, http://www.metacase.com, 2005.

[MOF] Meta Object Facility, http://www.omg.org/uml/, The Object Management Group, 2003.

[MV02] MarkV Homepage, http://www.markv.com/, MarkV Systems, 2002.

[NEI80] J.M. Neighbors, J.M., Software Construction using Components, PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1980, Technical Report UCI-ICS-TR160. Available at http://www.bayfronttechnologies.com/thesis.pdf.

[NEP00] Neptune – Check UML models and Generate Documentation, http://neptune.irit.fr/index1.html, NEPTUNE Consortium, 2000.

[OMG03] Unified Modeling Language Specification, version 1.5, http://www.omg.org/uml/, The Object Management Group, March 2003.

[PK02] R. Pohjonen and S. Kelly, Domain-Specific Modeling, *Dr. Dobb's Journal*, August 2002.

[POS05] Poseidon for UML, http://www.gentleware.com/index.php, Gentleware AG, 2005.

[ROS05] Rational Rose, http://www-306.ibm.com/software/rational/, IBM, 2005.

[SEL03] P. Selonen, Set Operations for the Unified Modeling Language, In P. Kilpeläinen and N. Päivinen, eds., *Proceedings of the 8th Symposium on Programming Languages and Tools (SPLST'03),* University of Kuopio, June 2003, pp. 70-81.
[SPI01] Diomidis Spinellis, Notable design patterns for domain specific languages, *Journal of Systems and Software*, 56(1):91–99, February 2001.
[SZI98] J. Sztipanovits, G. Karsai, and T. Bapty, Self-Adaptive Software for Signal Processing, *Communications of the ACM*, May 1998, pp. 66-73.
[TOG05] Together, http://www.borland.com/us/products/together/index.html, Borland, 2005.
[UMT05] UMT QVT home page, http://umt-qvt.sourceforge.net/, 2005.
[VIO05] Violet, http://www.horstmann.com/violet/, 2005.
[VUM05] Visual UML, http://www.visualuml.com/, Visual Object Modelers Inc., 2005.
[YAC03] Stephen C. Johnson, The Lex and Yacc Page, http://dinosaur.compilertools.net/, 2003.

Requirements for an Integrated Domain Specific Modeling, Modeling Language Development, and Execution Environment

T.D. Meijler

University of Groningen
t.d.meijler@rug.nl

Domain Specific Modeling Languages (DSMLs) offer powerful expressivity within clearly constrained domains. By using DSMLs domain experts can describe solutions in terms of models that make sense to them, and these models can be mapped to system realizations. Thus, DSMLs can move software development in these domains closer to the domain expert and make it more efficient. Large-scale use of DSMLs for realizing enterprise applications requires an extended infrastructure for developing and using these languages. In fact it requires an integrated modeling and execution environment in order to allow run-time adaptation of models and rapid incremental extension of the applications. In this paper we shall give an overview of requirements for an infrastructure that both supports developing and using DSMLs as well as executing the corresponding models as necessary for the Enterprise Application domain.

1. Introduction

Model-driven development (MDD) raises the level of abstraction for software development by expressing what a computer should do by means of (design-level) models that hide technical details. In MDD software applications are derived from models either through code generation or through interpretation. Applying a domain specific modeling language (DSML) instead of a generic modeling language such as UML promises to bring MDD even closer to the domain expert. DSMLs can make software development in specific domains more efficient since they close the gap between the expert and the software implementation.

Domain Specific Modeling Languages and their corresponding modeling environments –so-called Domain Specific Modeling Environments DSMEs– have already been widely used in industry, e.g., for workflow modeling [3], for modeling of chemical plants [12] and banking [5] etc [9].

Setting up such DSMEs has often been seen as too complex and requiring too much effort. It may involve building a special purpose editor, model storage and retrieval, and special purpose code generation, or any other means to give a model its run-time semantics (e.g., interpretation). Especially the latter aspect may be complex to realize. Various meta-modeling environments have therefore been developed that

support the development of such DSMEs. Most of these environments apply (UML-based) meta-modeling for realizing DSMEs. The best known are the Generic Modeling Environment (GME) [6], The Eclipse Modeling Framework (EMF) [2], and the Metacase environment [9].

The author of this paper has been involved in developing and using DSMLs for realizing large scale Enterprise Applications for Supply Chain Coordination[1] and the development of a meta-modeling environment to support this. As it turns out, in this domain the requirements for a meta-modeling environment are far beyond what current meta-modeling environments can offer. This is roughly due to the following:

- In contrast to most DSMEs, a DSME for realizing Enterprise Applications must allow the *integrated* use of *various* Modeling Languages, e.g. for modeling Workflow, Resources, Enterprise Structures, Product Models, Product Lifecycles etc.
- In contrast to most DSMEs, in Enterprise Applications there is the need for incremental extensibility of models and ad-hoc model adaptation. A large scale enterprise application will not be built in one go, and is never ready. It must be possible to add new business processes, new kinds of contracts, but it must also be possible to adapt business processes and contracts where needed, and even in an ad-hoc manner. Thus it is necessary to have a close integration between the Application and the DSME such that modeling and model execution can be interspersed.
- Due to the scope and size of such applications, it is no longer just one organization that is involved in building applications. In Enterprise Applications (so-called vertical) sub-domains can be distinguished, such as Construction, Banking, Health etc. Separate software-developing organizations have expertise in these sub-domains. In the context of DSMEs/ DSMLs this means that such developing organizations will extend the vocabulary of various DSMLs, e.g., with new kinds of contracts or new kinds of processes. These can than again be used by a subsequent software-developing organization (e.g. a consultant) to build customer specific applications. This also means that Enterprise Application development will be done in a "Software Supply Chain" [7].
- It must be possible to adapt an existing DSML to add or change modeling constructs. Moreover, it must be possible to add a new DSML to the existing set of DSMLs of the environment. The impact of changes of DSMLs on existing applications must be managed.

The contribution of this paper is to enumerate the requirements for a meta-modeling environment that supports the Enterprise Application domain. Due to the second and fourth bullet, the requirements in fact cover an integrated meta-modeling, modeling and execution environment. The paper also exemplifies the requirements. It does this mainly through a DSML for workflow modeling. The reason to use workflow as the central example is twofold: On the one hand, business process support forms the heart of Enterprise Applications. On the other hand, it is precisely in this horizontal sub-domain where the various aspects mentioned above can be exemplified.

[1] This has been done in a project called "Nucleus".

This paper is structured as follows: In section 2 we introduce the main Workflow modeling example. Also a second example is introduced, a DSML for lifecycle modeling, especially needed to exemplify the requirement for integration of DSMLs. In section 3 we first formulate a set of starting points and subsequently derive our requirements based on these starting points. In section 4 we discuss related work, and in section 5 we conclude.

2. Examples of DSMLs in the enterprise domain

2.1 Introduction

In this section we discuss workflow and lifecycle modeling as examples of DSMLs. These examples are used throughout this paper to introduce and explain the set of requirements. Both examples have been used and developed as part of our "Nucleus" project. The examples will only be discussed superficially: only little theoretical background will be given, mainly discussing how certain models look, and (as far as relevant) what they mean.

2.2 Workflow modeling

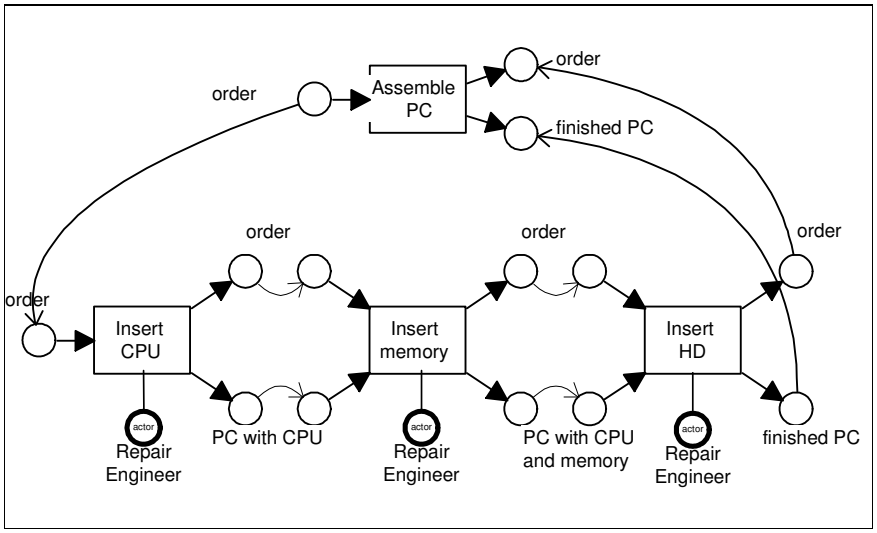


Figure 1 Modeled workflow example: Assemble PC activity

Figure 1 shows a modeled workflow activity: “Assemble PC”. The “Assemble PC” activity is a composite activity, consisting of three sub-activities: “Insert CPU”, “Insert Memory” and “Insert HD”. Activities have inputs and outputs, so-called “places” as visually modeled by open circles. The output places of one activity are linked with the input places of the next activity. E.g., the “order” output place of “Insert CPU” is linked with the “order” input place of “Insert memory”. Such links show how information is passed on (in the form of so-called “tokens”) from activity to activity. “Insert CPU”, “Insert Memory” and “Insert HD” also have a special kind of place on which an actor, who is responsible for the execution of the activity will be assigned.

For readers knowledgeable in workflow, it may be clear that this form of workflow modeling is related to Petri net modeling [1]. The description of this relationship lies outside the scope of this paper as are the pros and cons of this form of modeling.

Our activity modeling system uses activity *types*, such as “Assemble Activity Type”, or “Insert CPU Activity Type”. Activity types have two functions: The first one is as placeholder of the activity structure as given above for the purpose of instantiation: when an assemble activity must be executed, the “Assemble Activity Type” is requested to instantiate itself, and subsequently a copy of the structure shown in Figure 1 is executed. Instantiation can also be used to be able to introduce types of activities in different workflow contexts: e.g., a “Package PC Activity Type” may be instantiated to be used in this “Assemble PC” activity, but may also be instantiated to be used in a “Resell 2nd hand PC Activity” workflow. The second function of an activity type is describing the types of inputs and outputs: through the typing information (especially the types of tokens that can be passed on certain places), a visual modeling environment can infer whether two activities can be linked. For example, the “Insert CPU Activity Type” describes that an “Insert CPU Activity” produces an “Order Type” on its “order” output place, while “Insert Memory Activity Type” describes that an “Insert Memory” activity needs an “Order Type” in its “order” input place. Thus, these places may be linked.

2.3 Lifecycle Modeling

Figure 2 shows an example of a (somewhat simplified) lifecycle model of a product description, which is linked to a production process for such a product. In this model various lifecycle states (so-called life states) are described: “Initiated”, which is the start of the lifecycle; “Type Specification Defined”, which describes when a specification is complete; “Technical Implementation Defined”, when based on a specification a production process is also added to the product description; “Released for Test”, when the production process is in a testing mode, and “Approved”, when the product description has been approved including its production process. The arrows between the life states describe possible life state transitions, e.g., from the “Released For Test” life state, the life state can go back to the “Type Specification Defined” state, in order to adapt the production process. Note that once a product description has been approved, it can no longer be adapted, since there is no arrow back from the “Approved” life state. Life state transitions can incorporate checks whether the transition is indeed valid. For example, the life state transition from

"Initiated" to "Type Specification Defined" includes a test whether the specification is complete.

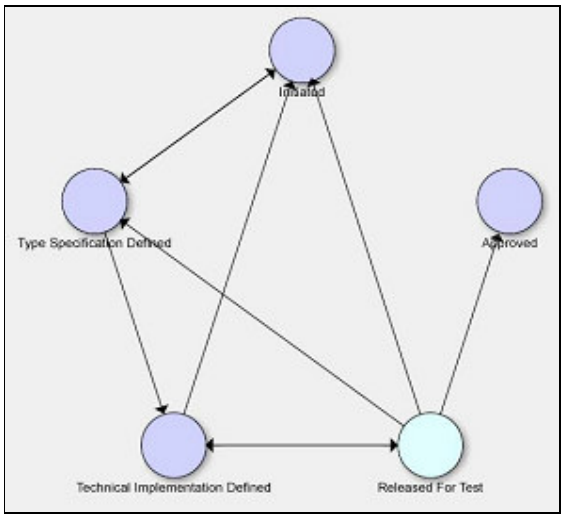


Figure 2 Example of a lifecycle model

2.4 What do these examples tell us?

This section introduced two examples of domain specific modeling. We can already see from these examples what we mean by a DSML, the fact that models in these DSMLs have a concrete meaning and thus map to (parts of) concrete systems, how models are presented, etc. Throughout this paper we shall see what other consequences there are in combining two or more DSMLs, how models must be adaptable etc. We shall refer to these examples in the rest of this paper to clarify our requirements. It should be noted that even though workflow modeling is the central example, this paper is not about workflow modeling. Moreover, these examples are not meant to show our overall vision on Enterprise Modeling; this is also outside the scope of this paper.

3. Requirements

We will now derive requirements for an integrated environment as based on a set of starting points, which are (largely) also requirements. These starting points have already been mentioned in section 1, but are reformulated more precisely. These are:

1. The development of DSMEs for DSMLs must be relatively cheap as is the starting point of all meta-modeling environments [6], [2], [9].

2. The Enterprise application domain implicates that it must not only be possible to support the development and use of different modeling languages, but also to support the integrated use of these modeling languages, since different modeling languages are used to describe different aspects of an application.
3. Support must be given for continuously changing requirements of users. This means support for changes of both the languages themselves as well as of the models and the corresponding applications. Such changes must have minimal, and certainly managed impact on running applications,
4. A software supply chain must be supported to allow different organizations to focus on certain parts of the software development process.

Our requirements will be illustrated using the examples given in section 2 above.

3.2 Enabling relative cheap DSME development

To make the realization of DSMEs relatively cheap as mentioned in point 1 above is a key requirement, that has already been realized by existing environments such as EMF and GME [2], [6]. This encompasses the following parts:

- The definition of a DSML. This includes the definition of the syntax of the DSML. It also includes (although others call this the static semantics, see the bullet on semantics) static constraints of DSMLs.
- The realization of a graphical editor for a DSML. In the workflow example, a graphical editor may represent a graph to a user such as shown in Figure 1, and allow the user to both to define and to adapt the workflow by inserting and linking activities.
- The realization of the semantics of a DSML. Realizing the semantics of a DSML means to give run-time meaning to a domain specific model. For example, what is the meaning of a workflow model, how is it executed?
- The storage and retrieval of a DSML.

EMF and GME [2], [6][8] apply meta-modeling for realizing all four aspects.

3.3 Supporting categories of DSMLs

Especially in enterprise applications, but presumably also in other domains, the integrated environment must be able to support various DSMLs as already indicated in point 2. This can, however, be further specified by understanding that there are two important categories of models. Modeling languages may support either one of these categories or both.

3.2.1 Prototypical Instance Models

The first category of Domain Specific Models is that of “prototypical instance models”. Prototypical instance models are models that are copied to be executed and thus seen as prototype for the run-time execution. For example, a workflow model (such as shown in Figure 1) is copied for each execution, such that each execution corresponds to a run-time instance. As another example, a product description

lifecycle model is linked to a product design. Each product design (design for a separate product) gets its own copy of this lifecycle model and traverses the states of this lifecycle model. Thus, in both workflow and lifecycle modeling prototypical instance models are used.

3.2.2 Type Models

The second category of models is that of type models. Type models don't describe directly –as in the former case– run-time executing instances, but describe generatively what kind of instance may occur at run-time. The activity types in our workflow example are type models, and thus the workflow example incorporates both prototypical instance models and type models. Other examples of type models are contract models, describing what kind of contracts are allowed in a certain organization, or product models describing what kind of product configurations are possible. UML class models are also examples of type models.

3.3 Integrating DSMLs

In point 2 above it was mentioned that to develop an enterprise application various modeling languages will be used to describe different aspects. Thus these languages must be integrated. In section 2 we gave the example of two different modeling languages that are both needed to model an application. In enterprise applications, all kinds of other modeling languages for describing resources, enterprise structures, contracts etc. are needed as well. Moreover, for realizing any application (also outside the enterprise application domain), separate modeling of the user interface will be needed.

The models defined in such different DSMLs must allow describing one integrated application. An integration of this form will be needed between our two example DSMLs. It must for example be possible to link the activity to approve the design of a product to a life state transition of a product model to the "Approved" state.

3.4 Integrating modeling and execution

Due to changing user requirements as mentioned above in point 3 above, models will be changed or added to adapt the applications correspondingly. This however must be done with minimal impact.

In "standard" model-driven development approaches, there is a radical separation between the CASE tool in which model-driven development takes place and the application in which execution takes place. As a result, changes or extensions to models can only be effectuated in the run-time environment by re-generating the application. (See also remarks of [11] on this subject). This is also referred to as the "big-bang" form of model-driven development.

In contrast, we require that extending models and changing models should not (in general) require such a big-bang approach: It must be possible to change models or extend models without having to re-generate and recompile the full application, and

(when possible) without having to stop the run-time use of the application. We thus require an integration of modeling and execution in one environment. Given our distinction between prototypical instance models and type models this requirement can be further subdivided.

3.4.1 Ad-hoc instance model adaptation

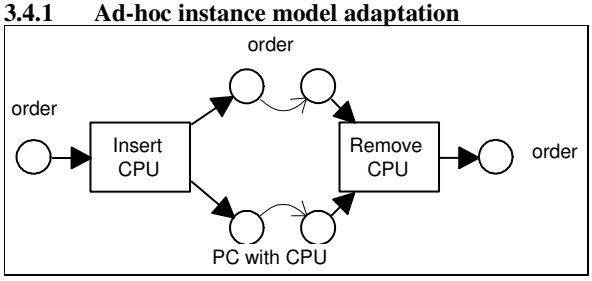


Figure 3 In case of a Cancellation after "Insert CPU" has taken place, "Remove CPU" is added to compensate for the activity that has already executed

In our workflow DSML example, ad-hoc adaptation of a workflow instance that is being executed is quite useful: executing workflow models may need ad-hoc adaptation in order to handle problematic circumstances. For example, in the Assemble PC process the order may be cancelled [14]. The workflow must than be changed drastically, such that the situation is handled smoothly: Assemblies that have already been made, and can be sold "as a whole" (e.g., often used housing/ motherboard combinations) must be stored and administrated for later use, other parts must be disassembled. Note that taking into account all such possible situations of error handling in the workflow definitions is almost impossible and leads to spaghetti models [14]. Instead, by allowing the workflow to be ad-hoc adapted such that the problem can be handled by an alternative sequence of activities is much better. Figure 3 shows an example. Such ad-hoc adaptation of prototypical instances requires that it must be possible to alternate between execution and modeling.

3.4.2 Ad-hoc type addition

Again in the workflow DSML an example can be found of ad-hoc type addition: It may be useful to promote a just adapted executing instance (as mentioned before) to a new type, so that it can be instantiated many times. The adapted workflow of Figure 3 is perhaps not a serious candidate, but in principle it is possible to promote this workflow instance as well: A type must be created e.g.: "Insert and Remove CPU Activity Type", and added to that type will be the prototypical instance as shown in Figure 3.

Note that ad-hoc type addition may be needed in many other domains, e.g., in the domain of inventory management, where it may be needed to introduce new inventory types in an ad-hoc way [9].

In [11] it is described how it should be possible to adapt types with an immediate impact on the run-time application. In the description of [11] this concerns adapting a UML class model. UML class models are interpreted. The result of an adaptation becomes immediately visible in the instances of that class model. [11] Uses this ad-

hoc type adaptation to allow for incrementally improving a UML model. The final UML model is then subsequently used to produce, through code generation, a stable application, which can no longer be adapted. In our work, adding and adapting types should remain possible, also for an existing application. This means that the danger exists that the application becomes unstable.

As a result, we require advanced, subtle mechanisms to adapt and add types, such that applications cannot become unstable. These subtle mechanisms must allow controlled change, similar as possible in, e.g., configuration management systems, i.e., changes only become effective in the running applications after quality control, moreover, users must be allowed to keep using older versions of types or sets of types in their applications. A detailed explanation lies outside the scope of this introductory paper.

3.5 Software supply chain support through Incremental Model-driven Development

Software supply chains (see point 4 above) are groups of software producing organizations that work together to produce final applications [7]. Software supply chains are based on software reuse: one organization produces a platform or component that the next organization can reuse to develop its component or application. That component may again be used by a subsequent organization, etc. Even the parameterization of software by a customer's support organization may be seen as a separate step preceding the final customer in such a supply chain.

As mentioned in section 1, catering for software supply chains for domain specific model-driven development is, in our point of view, very important in the domain of enterprise applications due to the scope and size of these applications. A software supply chain allows software developing organizations to focus on a certain expertise.

In many tools and approaches to model-driven software development (especially in "standard" MDA development tools) application software is generated from models as one whole and must also be compiled as one whole. This is an undesirable feature in a software supply chain. When software is generated and compiled as one whole, this means that each developing organization in the software supply chain has the responsibility of also generating and compiling code that has been developed by its predecessor. Moreover, this gives the developing organization access to that code, which may be unwanted from the point of view of intellectual property. Also, the whole software process of this developing organization becomes (too) tightly linked with the process of its predecessor: to adopt an update of the predecessor full generation and compilation must be followed, while often (when the predecessor only changes the implementation of its software) a recompilation of his own software will be sufficient.

Thus we require "incremental model-driven development":

Incremental model-driven development (IMDD) means to be able to build new software in a model-driven manner on top of software that has been realized previously in a model-driven manner as well. It is called incremental, since it allows incrementally adding layers on top of existing software.

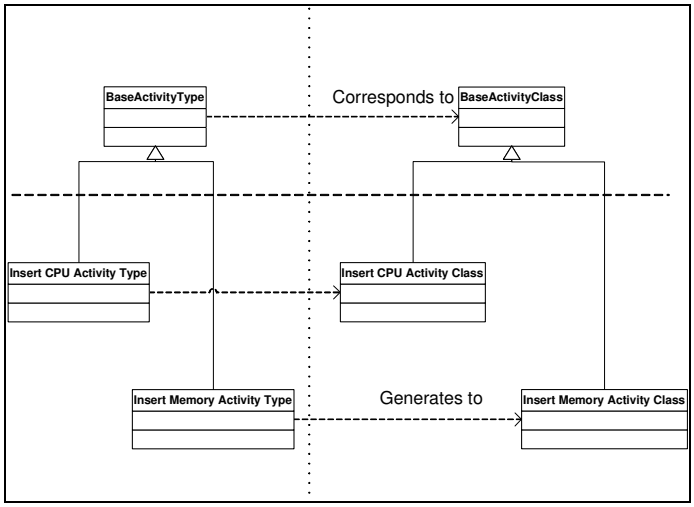


Figure 4 Illustrating incremental model-driven development

For example in workflow, incremental model-driven development means that one organization (A) has developed in a model-driven manner a workflow modeling language and a workflow activity execution engine. Another organization (B) extends on this.

Figure 4 shows how this might work in a simplified form. Organization (A) models "Base Activity Type" and generates from that "Base Activity Class". Through hand coding this base activity class is extended with code that takes care of the standard functionality of handling activity execution, e.g., checking that all places have a token, invoking a task etc. This is shown above the striped line. From left to right the difference is made between the models and the final implemented classes created from those models.

Organization (B) models specific activity types, e.g., "Insert CPU Activity Type" and "Insert Memory Activity Type" (not shown is specific type information, such as places methods etc.), these activity types all become subtypes of "Base Activity Type". Thus, organization (B) adds on to models already defined by (A). As a result, the generated classes will automatically be subclasses of the class "Base Activity Class" as implemented by (A).

Requiring IMDD in this case means that the subtype relationship from the (incrementally added) types of (B) to the "Base Activity Type" of (A), for which a class implementation already exists, will be mapped to a subclassing relationship with that class. Therefore the supply chain is supported, since (B) does not need to recompile this class, nor does (B) need access to the source code of that class.

3.6 Enabling impact management in model-driven development

As mentioned in point 3 above both models and DSMLs will change frequently. In a software supply chain different organizations reuse each other's model-driven

developed software as described in section 3.5. Therefore, managing the impact of these changes, which may now become inter-organizational, is essential. The forms that this can take are the following.

1. When the DSML definition (in fact, the DSML metamodel) changes, the domain specific models defined on basis of the old metamodel must be as faithful as possible converted to the new metamodel. Faithful in the sense that the intended semantics of the old domain specific model is preserved. As an example we have shown in Figure 5 the result of a workflow change that is needed when the workflow metamodel is changed for the purpose of exception handling [14]. The metamodel now incorporates so-called exit and cancellation activities. These can be added to a workflow definition for the purpose of exception handling. We note that in this case a new element in the modeling language is added, which can relatively easy be handled by adding the corresponding model parts. Other DSML changes will be more difficult to handle.

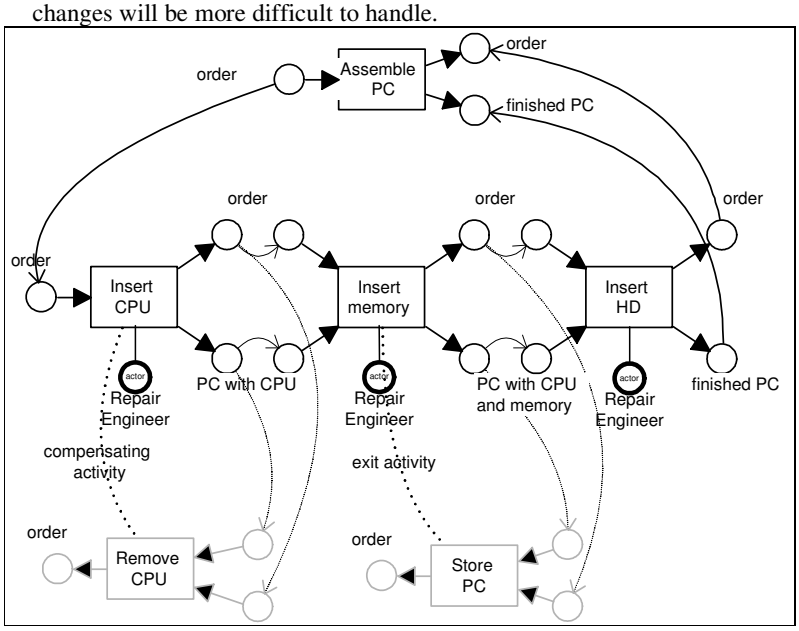


Figure 5 Assemble PC activity with compensating and exit activities added (grey)

2. When type models (see section 3.2.2), as defined in a certain DSML, change, its corresponding instances, e.g., specific workflow instances, may need to be converted faithfully.
3. Given the requirements of section 3.5, model driven developed code may re-use – either through specialization or though invocation– other model-driven developed code. When the reused models change, this may have impact on the dependent models. For instance, certain methods or properties may no longer be available, renamed etc. Impact management must take care that these changes are known and that the impact on the dependent models and code is made explicit. At least the developers of the dependent software will be informed of specific mismatches

between the old dependency and the update. At best certain changes are carried through automatically on the dependent models.

3.7 Offering an execution infrastructure

Given that we require an integrated modeling and execution environment, an infrastructure in which both can take place must be offered. This may incorporate aspects such as database access, both for instances –these are either data, or executing and prototypical instance models– as well as for other models, special forms of transaction management (e.g., optimistic transaction management for collaborative access to data and models), security, web services, user interface technology for access through internet browsers etc.

In the Appendix, Figure 6, a summary is given of starting points and requirements and their relationships as treated in this section.

4. Related work

Domain specific modeling and domain specific modeling environments are known from literature. Especially, GME [6][8] and EMF [2] are well known. Microsoft currently comes up with so-called Software Factories [4], and views domain specific modeling as its main approach to software development.

Our approach is with respect to its combination of requirements more ambitious than these other ones, due to its focus on Enterprise applications. Distinguishing aspects are:

- Its distinction between instance models and type models. GME and EMF do not give the possibility to have domain specific type models.
- Its integration between modeling and execution. For this kind of support, our work can be compared to work on adaptive data models [15].
- Its support for the software supply chain, and therefore on Incremental Model-driven Development. Even though incremental development is applied especially in object-oriented development, to the best of our knowledge, we have not seen it applied in model-driven development. See also our own paper [9] for further information.
- Its link between impact management and model-driven development. Impact management has already been applied in other forms of software development, e.g., OO development [13] and Database Schemas [10].

5. Conclusion

In this paper we have derived requirements for an environment that integrates the execution of domain specific models, the development of these models, and the development of the domain specific modeling languages. These requirements,

although quite ambitious, are necessary given the complexity of enterprise applications our application domain.

The ambition of our requirements as compared to other environments for the development of DSMEs is especially due to the ambition of our starting points:

- The integration of various different kinds of domain specific modeling languages, where we think "type modeling" is generally underestimated in current work on domain specific modeling.
- The integration of the run-time and the modeling world, in order to support (relative) ad-hoc adaptations and extensions of models, without having to completely recompile and restart a run-time system.
- The support for software supply chains, such that different organizations can focus on different expertise in the development of software. This supply chain support causes requirements for incremental model-driven development and impact management.

We have participated in realizing an environment called "Nucleus" for developing and using DSMLs for realizing enterprise applications for supply chain coordination. In this environment we have been able to fulfill many of the requirements, some of the aspects (e.g., impact management) have not been worked out in detail yet. Many of these requirements should become mainstream and part of the Model Driven Architecture, the current OMG standard for model-driven development.

References

[1] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT press, Cambridge, MA, USA, 2002

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. *Eclipse Modeling Framework*. Addison Wesley ISBN 0-13-142542-0, 2004

[3] Cosa: www.cosa.nl

[4] J. Greenfield, K. Short. *Software Factories Assembling Applications with Patterns Models, Frameworks and Tools*. Wiley ISBN –471-20284-3 2004

[5] V. Kulkarni and S. Reddy. *Model-driven development of enterprise applications* Compendium of Papers of the Industrial Papers Track of UML 2004

[6] A Ledeczi, M Maroti, A Bakay, G Karsai, J Garrett. *The Generic Modeling Environment*. In Proceedings of WISP, 2001

[7] D. G. Messerschmitt, C. Szyperski. *Software Ecosystems*. MIT ISBN 0-262-13432-2, 2003

[8] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, M. Maroti M. *On Metamodel Composition*. IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.

[9] T.D. Meijler. *Incremental MDA through Causal Connectedness*. UML Modeling Languages and Applications: UML 2004 Satellite Activities Lisbon, Portugal, October 11-15, 2004 Revised Selected Papers LNCS 3297 Eds. Nuno Jardim Nunes, Bran Selic, Alberto Silva, Ambrosio Toval

[9] Metacase. *Domain-Specific Modeling: 10 Times Faster Than UML*, whitepaper by MetaCase Consulting available at http://www.metacase.com/papers/index.html

[10] Y.G. Ra,, E.A. Rundensteiner, *OODB support for providing transparent schema changes*, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research table of contents [Multiple Versions]

[11] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorgbe, 2001. *The architecture of a UML virtual machine*. In Proceedings of OOPSLA'01. ACM Press, New York, 327–341
[12] Simtech: http://www.simtechnology.com/english/SimTech.php
[13] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, *Reuse Contracts: Managing the Evolution of Reusable Assets*, Proceedings of OOPSLA'96 ACM SIGPLAN Notices, Vol.31, No. 10, October 1996, pp. 268-285
[14] R. van Stiphout, T.D. Meijler, A. Aerts, D.Hammer, R. le Comte. *TREX: Workflow Transactions by means of Exceptions*. Proceedings EDBT Sworkshop on Workflow Management Systems 1998
[15] J.W. Yoder, F. Balaguer, and R. Johnson, *Architecture and Design of Adaptive Object-Models*. In ACM SIGPLAN Notices, 36(12), December 2001

Appendix: Overview of Starting Points and Requirements

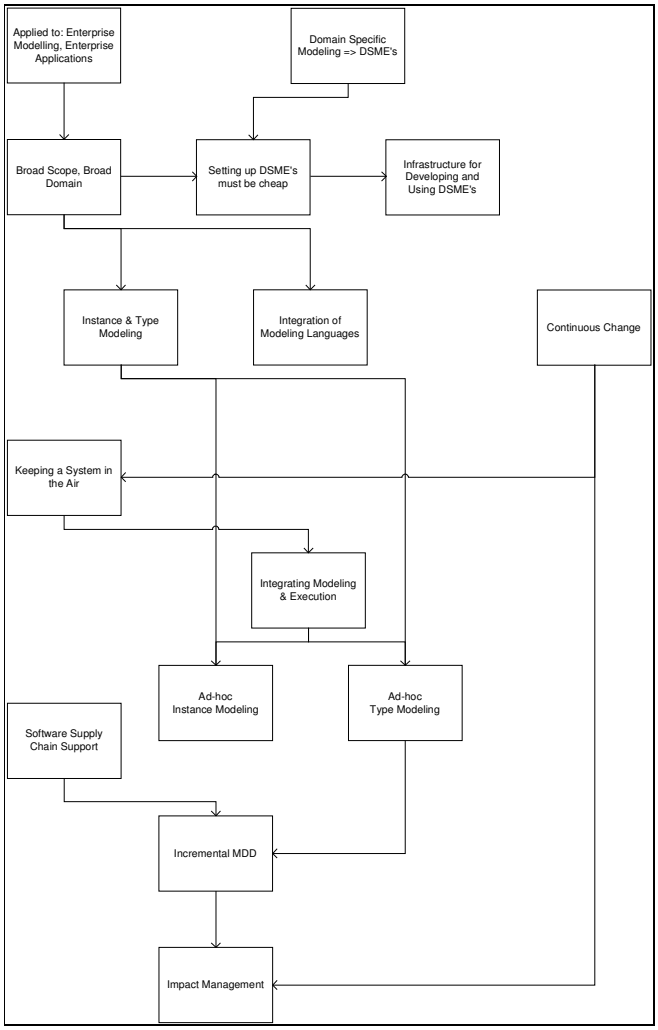


Figure 6 Overview of Starting points and derived requirements

Figure 6 gives an overview of starting points and derived requirements in the form of a dependency graph, each starting point or requirement is represented in a block. E.g., "Impact Management" is needed due to both "Continuous Change", and "IMDD" since by applying IMDD in the supply chain, models and their corresponding implementation are dependent of other models and their implementation. Note that the requirement "Keeping a System in the Air" even though referred to by "Continuous Change" is not really derived. This requirement is an ambition of the "Nucleus" project.

Moving towards Domain-Specific Modeling for Software Development

Zheying Zhang & Jyrki Nummenmaa

Department of Computer Sciences
University of Tampere, Tampere, Finland
{cszhzh, Jyrki}@cs.uta.fi

Abstract. Various observations support the view that domain-specific modeling (DSM) can improve productivity in software development. The typical explanation is that the resulting model is on a higher abstraction level and closer to problem domain concepts than models created with generic methods and using e.g. UML as the modeling language. In this paper, we elaborate on how the domain-specific modeling is on a higher abstraction level than a generic modeling and how this works as the basis for the better understandability. Meanwhile, in spite of the acknowledged benefits from using DSM, it has not yet become the prevalent approach in software development. It inspires us to further study the different strategies for organizations to move towards using DSM. This paper also discusses some of the strategies and their application.

1 Introduction

Most software development methods in use today support the development process with a similar recipe: conceptualize the problem environment where the desired software is put in use, specify solutions in a sketch by using the domain concepts, transform the conceptual solutions to diagrams that reflect the concepts and structure of a selected programming language, and finally generate the program code. In this process, the high-level domain solution is transformed to models that represent the implementation in code. As the low-level solution represents the technical detail, it is often hard to read and understand, and complicated to modify and extend. The transformation process tends to be time-consuming and error-prone [1]. In order to improve the process, the focus of software development has been shifting away from the technical implementation toward the concepts and semantics in problem domains [2]. Domain-specific modeling (DSM), therefore, becomes an appealing approach to software development. By using the DSM method, there is no extra transformation from the domain concepts to the implementation models. Instead, it specifies the solution directly using domain concepts from which the final implementation is generated [1]. The more directly application models can represent domain-specific concepts, the easier it becomes to specify application solutions.

The general-purpose methods, like the unified modeling language (UML), aim at solution specification in every necessary problem domain. Therefore, they are normally based on a low level of abstraction to describe the structure of an application

in terms of classes, methods, associations, dependencies, etc. and the interaction between objects. Unlike the generic modeling methods, a DSM method bases its concepts on the underlying problem domain concepts, and models the solutions using a higher level of abstraction than classes, methods and attributes. That is to say that a DSM method is constituted of domain concepts and their mapping to the technical implementation. Its users need not take extra effort in transforming domain solutions to the implementation diagrams, which makes it easy and fast to specify the problem solution, and further leads to improved software development process [1, 3, 4].

In spite of the acknowledged benefits from using DSM, it has not yet become the prevalent method in software development. Reasons behind it are mainly related to the insufficient understanding about the underlying principle of DSM and the lack of tool-based support. In this paper, we elaborate on how DSM raises the level of solution representation and how a higher level of abstraction supports the better understandability. Meanwhile, different strategies for moving towards DSM require further attention. Besides revealing the reasons behind the low acceptance of the DSM approach, we discuss strategies to develop the DSM and their application.

The remainder of the paper is organized as follows. The next section will clarify the fundamental concepts directly related to software modeling. Section 3 discusses the basic features of DSM by comparing it with UML and its profile. Section 4 further discusses the reasons of low acceptance of DSM in software industry, and suggests strategies to deploy DSM with an example of phone simulator interface development. Section 5 concludes the paper and discusses future research.

2. Basic Concepts

2.1 Software System, Method and Tool

Concepts involved in software development include software systems, software development methods, and the tools. Their dependencies can be captured in the model shown in Figure 1.

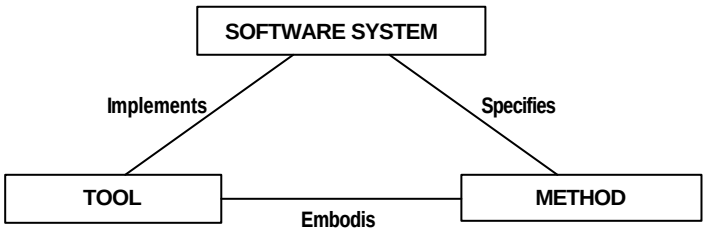


Figure 1 Support for software development (after [5])

A *software system* is a collection of components organized to fulfill a defined purpose [6]. It is the expected delivery resulting from a software development life cycle. As software systems become widely applied in our daily life, their development becomes market-driven. A quality software system that meets customers' needs and their

timing forms the ultimate goal of a software development project. To achieve such a goal, methods and tools form the essential support. A *method*, consisting of guidelines and rules of a specific underlying philosophy of software development [7], provides a means of defining problems, representing solutions, implementing and maintaining the software system. Different methods are applicable in different forms at different software development stages. Typical examples include interviews and questionnaires at the requirements stage, UML , OMT [8], and Yourdon's structured analysis and design [9] at the analysis and design stage, etc. A *tool* aids in accomplishing the software development activities [10]. It embodies a method to facilitate the process of solution specification and implementation. Therefore, different tools take a role at different development stages. Examples include the requirements engineering tools (e.g. Telelogic Doors [11]) for requirements documentation, modeling tools (e.g. Rational Rose [12], MetaEdit+ [13], etc.) for solution specification and management, programming tools (e.g. MS Visio Studio, Sun's JDK, etc.) for writing and compiling code, etc. Both methods and tools are indispensable to software development. Methods provide fundamental principles for solution specification and implementation, while tools enable and support the process. Without a proper tool support, it is hard to effectively deploy a method in the software development process. Therefore, besides the nature of a method, an available tool support forms an important criterion in method evaluation and selection for a software development project.

2.2 Software Development Methods

Due to the diversity of software applications and the continuing development of information technology, it is hard to define a universal method that suits every kind of software development. Therefore, a variety of methods and their supporting tools exist. It was estimated that there were over 1,000 brand name methods world wide in 1994 [14]. There is no doubt that methods have continued to proliferate.

As the number of methods grows, a straightforward question is to select a proper method in the method jungle. Surveys regarding methods have been carried out along with the evolvement of methods. The findings clearly show that a considerable number of organizations use in-house methods rather than commercial methods for software development [15-20]. The reason is twofold: the underlying principle of the methods is incongruent with their context of use [18, 21], and the tools lack flexible support to adapt methods that meet the requirements of the changing business environment. Because of the domain-specific nature of in-house methods, they are relatively easy to accept by the organizations.

In addition to the domain-specific methods, some generic modeling methods provide extension mechanisms to meet the needs of a particular domain. A typical example is the UML profile [22], which allows users to customize its notation to their particular domain or purpose by using stereotypes, tagged definitions and constraints [22]. Because the profile definition is an extension, there are no changes of the UML semantics. Although the domain concepts defined by the profile make the solution representation more domain-specific, there are no fundamental changes of the principle of the method. The complex semantics and the mapping from the high-level

conceptual solution to the low-level representation still exist. Besides, the limited tool-based support forms a barrier to deploy UML and its profiles in software development. In comparison with the complex semantics of the generic modeling methods, a DSM method is relatively simple and more easily acquainted by the software developer. It is a promising and attracting approach to model-driven development and worth further studying.

3. Domain-Specific Modeling Method

In order to discuss the principles of DSM and its features, we compare DSM against the generic modeling methods. As a point of comparison, we use methods, which use UML as their modeling language. Although these methods come in many variations, it is in fact more important for our discussion to specify the modeling abstraction level, and therefore it is quite sufficient to specify UML as the modeling language. Notably, our discussion carries over to other methods, whose modeling language is on the same abstraction level with UML. The discussion is made from the perspective of the abstraction level, the understandability, the tool-based support, and the support for systematic software development.

To provide a concrete comparison between DSM and UML-based methods, we use a running example of phone simulator interface development. We assume that a phone simulator consists of a set of buttons for user's input and a display for information presentation. The example includes the design of the simulator interface and the interaction with end users. MetaEdit+ [13], a metaCASE tool, facilitates the solution representation by using either the DSM method or the generic modeling method. The examples are shown in Figure 2 as a class diagram and in Figure 3 as a DSM model.

3.1 Abstraction Level

The abstraction level of a method indicates the level of details in solution representation. It forms the fundamental principle of a method. Generally speaking, the lower the abstraction level, the more detailed technical implementation it includes, and the more difficult it becomes to specify applications. Software development is the process to transform a high-level solution down to the low-level application code.

Models with intention of representing the problem domain are commonly on a high level of abstraction. Examples include the rich picture diagrams in software system methods [23] and different kinds of business process models. Models with intention of detailing solutions to a problem are normally located one level lower than models specifying the problem domain. They specify the problem solution based on the concept of implementation technique. Figure 2 shows an example of the design model, a class diagram specifying different types of phone buttons and their interaction with end users. It diagrammatically represents the structure of implementation of the phone buttons in Java. The diagram specifies classes and their inheritance relationships. The superclass *phoneButton* has two subclasses:

phoneNumberButton and *phoneControlButton*, which further have a set of inherited classes.

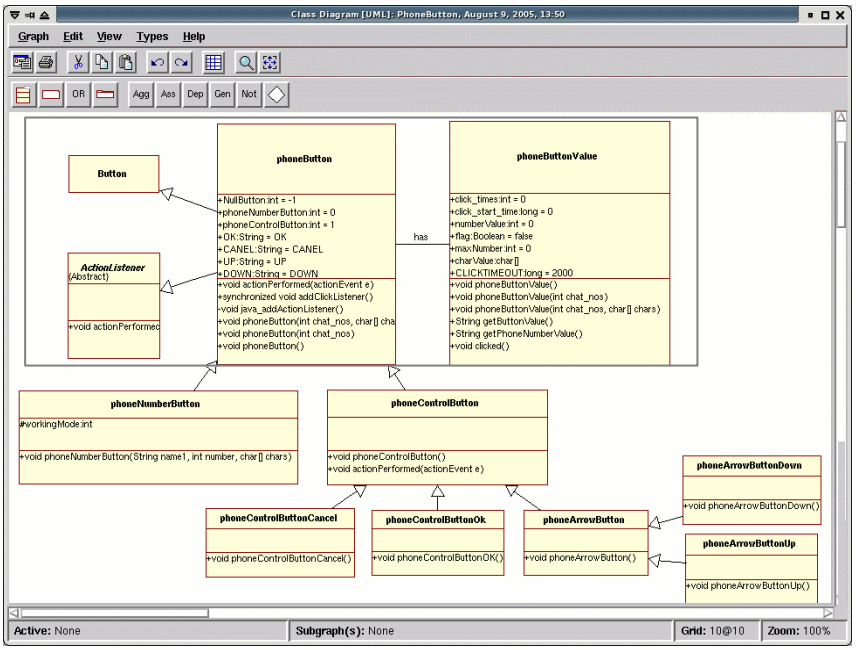


Figure 2 A class diagram: Phone Button

A basic interaction with user's input is to get a right button value when a user presses a phone button. In order to specify the solution, the class diagram includes the definition of a data structure to save values assigned to each button and the detection of the clicking time and the time-out on buttons. The detection is defined as methods in *phoneButtonValue* and shown below in the segment of code. Obviously, these specifications are not directly related to the features of the phone simulator. They are internal data and techniques related to the low level of abstraction, such as the implementation platform and the programming language. They vary in different development environment. When such a low-level abstraction is represented in design models, it is likely to confuse a designer.

```
   flag =false; // check time_out
   if (click_start_time == 0) {
         click_start_time = System.currentTimeMillis();
         click_times = 0;
         flag = false;
   }else
   if((System.currentTimeMillis()-
click_start_time)>CLICKTIMEOUT) { // Time-out
         click_times = 0;
         click_start_time = 0;
```

```
        flag = true;
}else {
        click_times++;
        if(click_times>=maxNumber)click_times = 0;
}
```

Unlike the class diagram, DSM methods are targeted to particular problem domains. Their concepts and structure conform to the domain concepts and rules. The inner working or the data for implementation, like the above example of the detection technique, is combined with the domain concepts and "invisible" to the designers. The rising abstraction level allows designers to concentrate on the required features of software products and shift their focus from technical implementation to design, which makes the modeling process natural and easy to carry out.

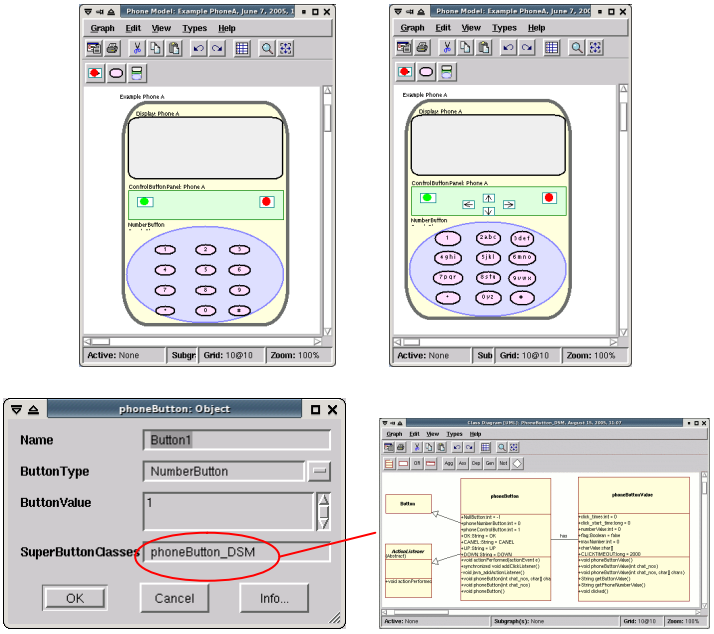


Figure 3 A DSM model: Phone simulator interface

As inner data for implementation is not directly related to the functionality of an application, it is often dependent of the development environment and can be specified and used as a common part in developing different applications within the same environment. Figure 3 shows examples of the phone simulator interface model using a DSM method. Instead of abstract concepts such as classes and relationships, the model is composed of three basic domain concepts: *Control Buttons* (rectangle buttons with different symbols inside), *Number Buttons* (oval buttons with values inside), and *Panels* (round-edged rectangles). The low-level specification of buttons, such as classes *Button*, *ActionListener*, *phoneButton* and *phoneButtonValue*, are pre-specified and hidden in the code generator of the model, as shown in the right-corner

screenshot in Figure 3. Developers have to specify different control buttons, number buttons, and the corresponding values when designing different phone simulators. An example to specify number button 1 is shown in the left-bottom of Figure 3. Comparing the DSM models with the class diagram, we can see classes in the shadow in Figure 2 become the common but " invisible" specification in the DSM model, and the rest of the classes become the instances of the object type: *Control Button* and *Number Button* in the DSM model. In this case, raising the abstraction level is based on hiding some of the information of the model of the implementation, or equally, it can be seen as just choosing certain elements from the implementation model.

Raising the abstraction level, however, is not just based on hiding elements. Consider that the GUI package implements operations for checking if a phone control button is up or down and operations for reading the position of the cursor. A programmer may implement a routine for checking if a button was selected. This would include a series of operations reading the button and coordinate information and comparing this with the button placement. On a higher abstraction model, it would be possible to just introduce a method for checking if a button is selected (and to hide the implementation details from the programmer) or just to introduce the button with a facility to connect a callback function to the selection of the button. In this way, we introduce new, higher level elements, which use the services of the old elements but operate on a more suitable abstraction level.

Therefore, we claim that the raising of abstraction level in DSM comes from exactly two sources: hiding information and introducing new concepts on a higher abstraction level.

3.2 Understandability

The understandability of the model is discussed from two perspectives: the modeling language and its instance, the actual model built using the language.

Considering the modeling language, because a DSM method conforms to the application domain with which the developer is familiar, it is easier to understand than the generic methods. Take an example of the model in Figure 3, *Number Button* is a common concept in phone interface design, and it is obvious that the phone panel contains twelve number buttons – twelve instances of the object *Number Button*.

The concepts and notations of UML, however, are related to the solution implementation. Like the example in Figure 2, the model specifies the phone button as a set of classes and relationships. Some are abstract, such as ActionListener which is not directly related to the concepts in the application domain but to the solution implementation, i.e. they visualize the code. In order to represent the solution, a developer has to understand the underlying concepts defined in UML and map the problem solution to the concepts in UML, while a DSM method user can model the solution directly using the domain concepts. Obviously, it sets higher requirements for developers to understand and use UML than DSM, and distracts developers from issues related to software modeling to topics applicable to low-level implementation at the software design stage. For a highly experienced developer, this may not be a problem at all, but for a mediocre developer or a novice, the task can be time-

consuming and error-prone. Also, the gained knowledge of the internals of the package is not necessarily anything worthwhile.

Meanwhile, like most comprehensive standards, UML is rather large. If it is used in a " rich" way, the necessary modeling elements and thereby also the model may be hard to comprehend. Arguably, a project typically only uses a limited and well-chosen set of modeling primitives from UML, thus reducing the number of features of UML necessary for understanding the model. Also, when moving from one application area to another, the modeling concepts do not really change as long as the set of modeling constructs picked from the whole UML is the same.

In order to explain an improved understandability of DSM models, we move our attention to the actual models. As we know, by hiding information and introducing the new higher-level domain concepts, DSM raises the abstraction level closer to the problem domain (i.e. analysis and design), as opposed to the solution domain (i.e. implementation and coding). The concept of information hiding is not an innovation of DSM. It has been introduced and applied more than thirty years before [24]. However, it is notable that just blindly or randomly hiding some information and introducing new conceptual elements is unlikely to make a model understandable. The key issue of information hiding in DSM is that the methods are *purpose-built*. The method contains exactly the concepts that are common and important for application development in a given domain. These essential concepts are not applicable in another domain and can not be directly represented by the generic methods.

3.3 Tool-based Support

As discussed in section 2, tools incorporate methods. They enable and support the software development activities. Any modeling methods, to be useful, must be incorporated into the CASE tool.

UML is widely supported by CASE tools. Examples include commercial tools such as IBM's Rational Rose, Microsoft Visio, etc. and open source tools such as Umbrello UML Modeller, FUJABA Tool Suite, DIA, etc. These tools provide diverse ways to support software modeling, from a simple diagram drawing tool to a comprehensive tool featuring documentation, code generation, reverse engineering, synchronization between code and diagrams, etc. Besides a large amount of tools with diverse functionality to support generic methods, we notice that these tools support an all-purpose modeling language and offer fixed code generators that try to fit all situations [25]. That is to say the methods are hard codified in tools, and difficult to change or customize [26, 27]. The standard UML profiles are incorporated by some CASE tools. However, the number of standard profiles is small, and profiles defined by organizations can not be used because the majority of above mentioned tools do not incorporate them.

Generating full code from high-level abstraction is an issue concerning tool support and domain expertise [25]. Besides above mentioned features of CASE tools, the DSM tools emphasize full code generation and flexible construction and adaptation of the modeling methods. Therefore, instead of a modeling tool, the DSM tool allows both the design of DSM language and code generators separately by the

domain-experts so that both fit the requirements the situation imposes [25]. Due to the additional requirements of the DSM tool, there are not so many DSM tools as the UML tools. MetaEdit+ [13] is an example, which provides an integrated environment for method specification, software modeling, and full code generation. More recently, open and customizable modeling environments, like Eclipse EMF & GEF [28], have appeared, supporting DSM [25].

3.4 Supports for Systematic Software Development

DSM methods reflect the concepts and rules in an application domain, underlying which is the idea that the similar problems can have similar solutions in the same development environment. This improves systematic reuse practice in software development, such as reusing domain concepts and rules, design patterns, software architecture models, requirements, tests, and many other interim products produced in the software development life cycle. Therefore, DSM guides the standardization of products and processes based on commonalities in a set of similar software products. It is beneficial to deploy the DSM methods in product family development [2].

In practice, by deploying DSM, the organization can divide the work in an effective manner. As experts and experienced developers have sound knowledge about the application domain and its product development, they work on the method engineering level to build and maintain the domain-specific methods and their components. The novice and mediocre developers work on the product level to specify the functionality of software products by using the methods and providing feedback about the method usage.

4. Strategies to Deploy DSM methods

Although DSM is a promising approach to representing and implementing domain-specific concepts, deployment of DSM methods has not become the phenomenon in software industry. Because every method has its strengths and weaknesses in a certain development context, we cannot evaluate it apart from understanding what the objective is and how the organization will use it to realize the objective. The organizational development environment forms an important factor in relation to the usage of DSM methods [24]. In the following, we further discuss it from the perspective of the organizational strategies, the organizational support, and the technical support.

Organizational strategy for software development – The DSM methods accelerate the development process in organizations whose strategy is to develop a family of products. Due to the great commonality of products within the same application domain, organizations can accumulate knowledge and experience in their previous development projects, which makes it easy for them to abstract domain concepts and rules, and to deploy the DSM methods. Meanwhile, if applications in a given domain are developed by following the same DSM method, systematic reuse can be achieved across different development projects, which improves the efficiency and effectiveness the product family development. However, if the organization aims

at the project-based development, the development activities are not limited to a given application domain. It becomes difficult to invest on the DSM method for every individual development projects.

Organizational support of the new method deployment - Some empirical studies [17, 24] show interesting evidence for that the most widely used techniques in software development date back to techniques from the pre-structured and structured eras. The software industry is notoriously slow and reluctant to accept new techniques [17, 29]. Because organizations get used to their development approaches and the environment, introducing a new method need much additional attention to the supporting tools, the data transformation, the staff training, etc. Therefore, instead of a standalone activity, deployment of DSM methods is a part of an organization's overall process improvement strategy and requires years of investment before it pays off. It needs a clear management vision and commitment to introduce and sustain DSM methods.

Technical support – Technical support drives successful deployment of methods. DSM methods can be deployed in several well-established domains, such as the domain of graphical user interfaces, while they are not well deployed outside the well-established domains. Besides the lack of knowledge about the application domain, existing tools rarely provide a proper metamodeling language and the associated facilities that flexibly support developers to specify and implement software development methods according to their needs. Therefore, due to the immature status of the DSM method, and the insufficient tool-based support, it is too early for organizations to adapt it. In addition, the competence of method developers forms an additional factor that affect the deployment of DSM.

Apparently, the different strategies for moving into DSM require further attention. In the following sections, we discuss some of these strategies.

4.1 Greenfield Development

Greenfield development provides a strategy to develop domain-specific methods from scratch. It is mostly used to develop methods in an emerging application domain where it is hard for developers to deploy a generic modeling method for solution specification. Examples of the application domain include diverse applications in E-business or embedded software [30].

According to the method development process model [18], a domain-specific method is developed by method engineers who are familiar with the application domain. Different from the generic modeling methods, the DSM method focuses on identifying the domain concepts and their structure. In general, the commonalities of applications within the same domain are abstracted as domain concepts and defined in the method, while the variables are defined as concepts with an interface to assign value to variation parameters. The domain rules and the mapping between domain concepts and the programming code are encoded into the conceptual structure of domain models.

It is worth noting that the DSM development is an iterative and incremental process. As the experience and knowledge of the application domain is accumulated, more requirements are raised to improve the DSM methods. Therefore, DSM is a

long-term program of an organization. The organization's maturity level increases along with the improvement of DSM.

4.2 Transforming Generic Models to Domain-Specific Models

In most cases, DSM methods are developed according to existing modeling methods. Similar to the greenfield development approach, the key issue is to identify the domain concepts and rules out of the existing models. However, the transformation from a generic method to a domain-specific method is a process to extend the semantics and notations of existing methods to specify the underlying concepts of a problem domain.

For simplicity, assume that the application has been modeled using UML, and the model includes items such as packages, interfaces, classes, etc. Typically, before developing the domain-specific methods, some experience of developing applications for the problem domain has already been gained, i.e. the method engineers know the needs of the application development. Even if there are uncertainties, it is better that the most experienced developers solve them in advance. In large, the process may go as follows.

1) Identify the immediately useful elements. E.g. in the GUI example a method to add items to a pull-down menu might be such an element.
2) Identify the necessary elements that could not be directly picked from the UML model.
3) Design the packaging by extending the notation of the UML. Notice that elements from Step 1 and Step 2 could be put in the same concept in the DSM.
4) Finalize the implementation of the DSM model for the necessary part. Notice that here it is also possible to model the implementation in the original UML model.
5) Verify the model with some example implementation.

Different development groups within the same organization may need different DSM models, which may be based on the same UML model. Instead of a general-purpose method, the essential of DSM methods is to cater for a specific problem domain. Therefore, there is no need to model them all in a general good-for-all model, because they may be overlapping but not equally the same.

5. Conclusion

Software application development is market-driven. Developing a quality application to meet stakeholders' needs requires development methods to fit into the development environment. Given that the development methods are easy to construct and thus modify with a proper tool-based support, it is important to provide a modeling method designed specifically to aid in software development within the application domain.

This paper elaborates on the philosophy and principles of the DSM. It clarifies that raising abstraction level for solution representation is the essential feature of DSM, which is realized by generating new high-level domain concept and hiding detailed and implementation-specific information. This leads to improved understandability and systematic application development. In addition, it also improves the maturity of the software development process with more consistency and quality.

The discussion helps organizations to understand the principle of abstraction level rising, and the approaches to realizing it. The paper briefly suggests strategies to deploy the DSM for application development within an organization. Future research on how to apply DSM method to improve the maturity of software development process forms an emerging topic in this area. In addition, the relationships between DSM and other software development paradigms, such as software reuse and OMG's model driven architecture (MDA), are all areas worth exploring.

References

1. MetaCASE, Domain Specific Modelling: 10 Times Faster Than UML. 2000, MetaCASE Consulting.
2. Wada, H., et al. A Model Transformation Framework for Domain Specific Languages: An Approach Using UML and Attribute-Oriented Programming. in Proc. of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics. 2005. Orlando, FL, USA.
3. Wile, D. Lessons learned from real DSL experiments. in Proc. of the 36th Hawaii International Conference on System Sciences. 2003.
4. Wegener, H. Balancing Simplicity and Expressiveness: Designing Domain-Specific Models for the Reinsurance Industry. in Proc. if the 4th OOPSLA Workshop on Domain-Specific Modeling. 2004.
5. Zhang, Z., Model Component Reuse - Conceptual foundations and application in the metamodeling-based systems analysis and design environment, in Jyväskylä Studies in Computing. 2004, University of Jyväskylä: Jyväskylä. p. 76.
6. Sage, A.P., Systems Management for Information Technology and Software Engineering. 1995: John Wiley & Sons.
7. Wynekoop, J.L. and N.L. Russo, System Development Methodologies: Unanswered Questions and the Research-Practice Gap. Journal of Information Technology, 1995. 10: p. 65 - 73.
8. Rumbaugh, J., et al., Object Oriented Modeling and Designing. 1991, Englewood Cliffs New Jersey: Prentice-Hall.
9. Yourdon, E., Modern Structured Analysis. 1989, Englewood Cliffs, New Jersey: Prentice-Hall.
10. Lyytinen, K., K. Smolander, and V.-P. Tahvanainen. Modelling CASE environments in systems development. in Procs. of the first international conference on advanced Information System Engineering. 1989. Kista, Sweden.
11. Telelogic, Telelogic Doors. 2005. http://www.telelogic.com/products/doorsers/doors/
12. Rational, Visual Modeling with Rational Rose. http://www.rational.com/products/rose/index.jsp?SMSESSION=NO, IBM Corporation.
13. Kelly, S., K. Lyytinen, and M. Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. in Proceedings of the 8th International Conference CAISE'96. 1996: Springer-Verlag.
14. Avison, D.E. and G. Fitzgerald, eds. Information Systems Development: Methodologies, Techniques and Tools. Information System Series. 1995, McGraw-Hill Book Company.

15. Fitzgerald, B., The Use of Systems Development Methodologies in Practice: a Field Study. Information Systems Journal, 1997(7): p. 201 - 212.
16. Russo, N.L., R. Hightower, and J.M. Pearson. The Failure of Methodologies to Meet the Needs of Current Development Environments. in Proceedings of the British Computer Society's Annual Conference on Information System Methodologies. 1996.
17. Barry, C. and M. Lang, A comparison of 'traditional' and multimedia information systems development practices. Information and Software Technology, 2003. 45(4): p. 217-227.
18. Tolvanen, J.-P., Incremental Method Engineering with Modeling Tools: Theoretical principles and Empirical Evidence, in Department of Computer Science and Information Systems. 1998, University of Jyväskylä: Jyväskylä.
19. Hardy, C.J., J.B. Thompson, and H.M. Edwards, The Use, Limitations, and Customisation of Structured Systems Development Methods in the United Kingdom. Information and Software Technology, 1995. 37(9): p. 467 - 477.
20. Necco, C.R., C.L. Gordon, and N.W. Tsai, Systems analysis and design: current practices. MIS Quarterly, 1987. 11: p. 461–476.
21. Lang, M. Hypermedia systems development: Do we really need new methods? in Proceedings of the Informing Science + IT Education Conference. 2002. Cork, Ireland: InformingScience.org.
22. OMG, Catalog of OMG Modeling and Metadata Specifications. 2005, Object management Group, Inc.
23. Checkland, P. and J. Scholes, Soft Systems Methodology in Action. 1990, Chichester: Wiley.
24. Fitzgerald, B., Systems development methodologies: the problem of tenses. Information Technology & People, 2000. 13(3): p. 174-185.
25. Iseger, M., Domain-specific modeling for generative software development. 2005, http://www.ITarchitect.co.uk.
26. Hofstede, A.H.M.t. and T.F. Verhoef, Meta-CASE: Is the game worth the candle. Information System Journal, 1996(6): p. 41 - 68.
27. Kelly, S., Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+, in Dept. of Computer Science and Information Systems. 1997, Jyväskylä University: Jyväskylä.
28. Kelly, S., Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. 2004.
29. Gibbs, W.W., Software's Chronic Crisis. Scientific American, 1994. 271(3, September): p. 86 - 95.
30. DSM, DSM Case Studies and Examples. 2004.

The MICAS Tool

Johan Lilius[1], Tomas Lillqvist[1], Torbjörn Lundkvist[1], Ian Oliver[2], Ivan Porres[1], Kim Sandström[2], Glenn Sveholm[1], and Asim Pervez Zaka[1]

[1] Department of Computer Science, Åbo Akademi University, Finland
[2] Nokia Research Center, Helsinki, Finland

Abstract. The MICAS architecture is a novel SoC architecture for SoC design for mobile phone peripherals. Characteristic for MICAS is the separation of data-flows from control-flows. To design MICAS "modules" we have implemented a tool that is based on UML and MDA. The tool makes it possible to visually specify a MICAS system by dragging and dropping modules, busses, and connections onto a canvas. The design can then be transformed into a SystemC simulator for further testing. The MICAS tool is implemented using CORAL. CORAL is a metamodel-independent software platform to create, edit and transform new models and metamodels at run-time. CORAL provides support for a number of languages, including UML and MOF.

1 Introduction

The move to developing systems from high-level models and generating the code and intermediate model automatically is generally known as Model Based Development (MBD) [13] of which the OMG's Model Driven Architecture (MDA) is one implementation. While the concepts of MBD are not new in principle, we are now at a point where the technologies to support the introduction of these engineering principles into system development is possible; standard languages such as UML, representation formats such as MOF and XMI have been created. The increase in abstraction afforded by these techniques means that the engineer can now concentrate more on "what" the system is rather than on "how" the system is to do it. MBD can be characterized by diagram in figure 1.

The basic ingredients are models, languages and platforms. The basic development step in a model based development method *transforms models* - written in some language such as some profile of UML - into more detailed models by *architecting* the model onto a particular *platform*. In MDA terminology the source model of this transformation is known as a *platform independent model* (PIM) and the target model as a *platform specific model* (PSM). The distinction between a language and a platform is sometimes difficult to make because a language often implies a certain platform (c.f. Java and J2EE). In the context of this paper we shall keep this notions separate. A language is understood as a subset of UML specified by a profile. A platform is a collection of elements, such as available compilers, frameworks, architectures and even properties such as whether certain types of communication are synchronous, asynchronous (or

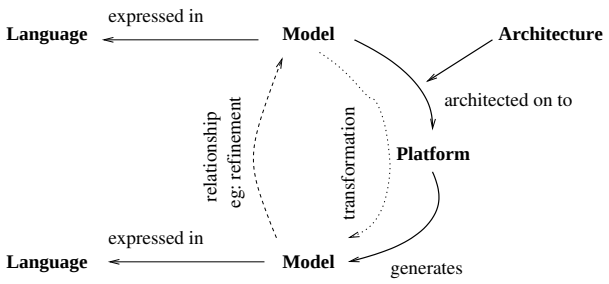


Fig. 1. The basic structure of Model Based Development

even "don't care" in very generic places). The stricter the properties means that the platform is more concrete (less abstract) and implementation oriented.

We distinguish between the following kinds of transformations:

- A *transaction* is the most restricted form of transformation. In a transaction neither the language, nor the platform changes. Typical transactions are normal editing of a model i.e. addition of class or objects etc.
- A *translation* keeps the platform constant, but changes the language. We have explored the translation of SA/RT models into UML models in [15].
- A *transformation* allows both the platform and the language to change, although the language of the source diagram must be able to express the concepts required by the platform. A transformation is characterized by the change of level of abstraction or platform and thus the move of a design towards its implementation. A typical transformation would be code generation.

The transformations are supposed to be automatic in nature but necessity in the amount of information required for a fully automatic transformation dictates some manual components - these however should be expressed through the architecture.

Much MBD work has focused on single platforms (notably Java based frameworks) with monolithic architectures (c.f. [3]). The longterm goal of our work is to develop a model-based development method for Hardware/Software systems[3]. In such a context we may have a number of platforms onto which a model could be potentially mapped and for each platform many ways in which that model could be architected onto that platform.

This paper discusses the first version of a tool that we are developing whose aim is to help explore different ways that a model (i.e. an application) could be architected onto a platform and thus help us in our long-term goal. The tool can be seen as a case study in MBD where we try to automate as much as possible of the process of architecting. The goals of this tool development are among others:

[3] An overview of our ideas applied to protocol processor application design can be found in [7]

1. To act as a case study in MBD, and to help us understand what parts of the process of architecting can be automated. At the same time we will also gain deeper insights into how a platform should be defined in an MBD process.
2. To act as a driver for the development of the CORAL [14] framework. Several features, e.g. the transformation engine and the model management support, have been designed and implemented because the need arose in the context of this tool development.

The version of the tool presented in this paper is developed to help us architect applications onto the MICAS platform (c.f. section 2 below). MICAS is a novel hardware platform that is developed at NOKIA. It intended as an implementation platform for applications with high data-streaming requirements, where also the sources and sinks of theses data-streams may change dynamically during the execution.

The tool consists of a number of components. The basic interface is a diagram editor (see figure 3) that is used to design a specific instances of the MICAS platform onto which the application will be mapped. This instance is represented as model in a UML-profile, the MICAS profile. The tool also implements a number of transactions (c.f. above) for the MICAS models that add more detail to the model. The final result of this series of transactions is a model that can easily transformed into SystemC. The SystemC code can then be compiled and linked to obtain a simulation model of the application.

The structure of this paper is as follows. We will first discuss the general ideas behind the MICAS architecture and present the MICAS simulation framework. Then we will present the MICAS editor and discuss the transformations implemented in the tool. Finally we will close off with some conclusions and discussion of future work.

2 The MICAS Platform

The aim of this section is to motivate and explain the ideas behind the MICAS architecture. Our basic design assumption is that we have a considerably large set of hardware processes that process a considerable amount of data. These processes are referred to as modules. Modules may input and output data which at an abstract level can be considered data streams. The main goal when developing the MICAS platform has been to address designs where these streams have a great variation over time in where from and where to they are routed. Such applications are typically complex multimedia applications like, personal video recorders (PVR), media streaming stations etc.

It's often problematic to design an efficient architecture for silicon process communications since the quality of a particular design layout is determined by applications and application loads. A number of typical solutions exist to design silicon architectures capable of flexible inter-process data communication. One solution is the universally connected network [5], which could be implemented as an all connected switching matrix or a universally connective bus. The problem with such a solution is that it does not scale when the number of communicating

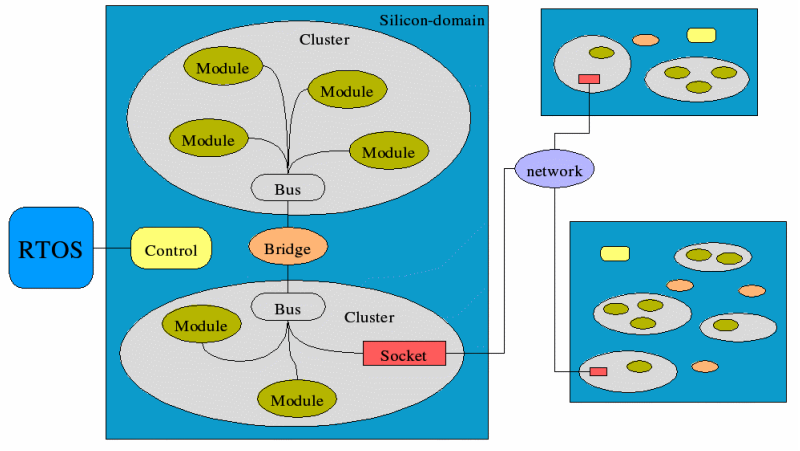


Fig. 2. An overview of the MICAS platform

processes grow. One single resource will handle all inter-process communication and it will either become a bottleneck or unreasonably complex.

Network on chip solutions, more specifically lattice or grid network architectures [9, 8, 10, 12], implement data transfer in a network of nodes where each node may be a hardware process or a module in he meaning described above. The streaming data is transfered as packets from node to node in a network where routing decisions are typically made in each node. In our opinion this kind of a solution is flexible and scalable but it has several drawbacks. The transfer delay is considerable because of node delay. Routing in a grid network is sometimes computationally complex, and the routing algorithms are often complicated and unreliable. The routing performance is often verified by statistical means and testing, and their performance is prone to the risk of congestion and deadlocks in untypical traffic situation.

Cascading architectures or streaming architectures [11] lend from the design of ALUs. The communicating processes are divided into layers that are unidirectionally and fully connected to the next layer of processes. The architecture facilitates a cascading flow of data between these successive layers of processes. Such an architecture is computationally very efficient. It's very fast with a small delay while still using a quite modest amount of silicon resources. The design methodology for a cascading architecture becomes difficult if the number of processes and the number of possible data streaming applications grows large. A consequence of necessary restrictions made during design is a lack of flexibility and a strict dependence to given applications.

We propose to use a combination of a universally connected network and a grid network based solutions. We call the resulting platform MICAS. An overview of its structure is given in figure 2. The communicating processes, called *modules* in MICAS - are divided into *clusters* that are universally interconnected via a

bus. Buses in turn are connected to each other via *bridges*, as nodes in a grid network. The MICAS platform forms a clustered networking architecture. One domain is designated the master domain. It will be controlled from the outside of the MICAS system, in the picture by a RTOS. In practice this could be an interface to the Symbian operating system on a mobile phone.

The network can be plain wires, bluetooth, or WLAN for example. The idea is that the exact communication between silicon-domains is encapsulated into a *socket*, which is the connector between two domains. Presently the exact structure of a socket is still open, so for the examples used in this project we have assumed that the socket is just a plain electrical connection using wires. The final version of the socket will include a protocol for detecting the connection and disconnection of a domain.

Since communication between processes is dynamic in terms of change in actual source destination pairs, there is a need for controlling and configuring the connections. There is also a need to setup and control the modules. Some architectures, especially packet based architectures like the grid network solution, have an intimate dependency between the application data and its control. In such a system routing data is included in each routed data packet. This implies that knowledge of routing resides in the source of data and that routing decisions are distributed to the nodes.

On the other hand a stream based control gives the possibility to control data transfer separately from a third entity, separating data flow from control flow. In the MICAS paradigm we have chosen to let all data communication be configured and controlled from a central unit - a controller. Behavior, any routing or other dynamic behavior of the network is centrally located. The advantage is that any future updates or changes in dynamic behavior is easily done in one part of the design. An additional advantage is that the dynamic behavior can be implemented in a programmable device such as a FPGA or a microprocessor. Updates can be done in the field without updating the silicon

The controlling device serves at the same time as a command interface to any external entity. It defines what services the domain is providing. Note that the set of services may be dependent on what slave domains are connected to the master domain (e.g. a printing service is only available if a printing domain is connected to the master domain). From a programmers point of view this command interface acts as a hardware abstraction layer.

The canonical example we will use in the rest of the paper consists of a simple application that streams picture data from a camera through an image manipulator that adds a watermark to the picture to a screen. We will also use this example to explain the programming interface of MICAS modules. A UML diagram using the MICAS profile that shows the structure of this example is given in figure 3. The diagram is a refinement of the UML object diagram, where we have created custom pictorial elements for all the MICAS stereotypes in the profile. At the moment we have one diagram that contains all the domains, i.e. in the picture there are 2 domains. In the future we will add a separate domain interconnection diagram to show the high-level structure of the system.

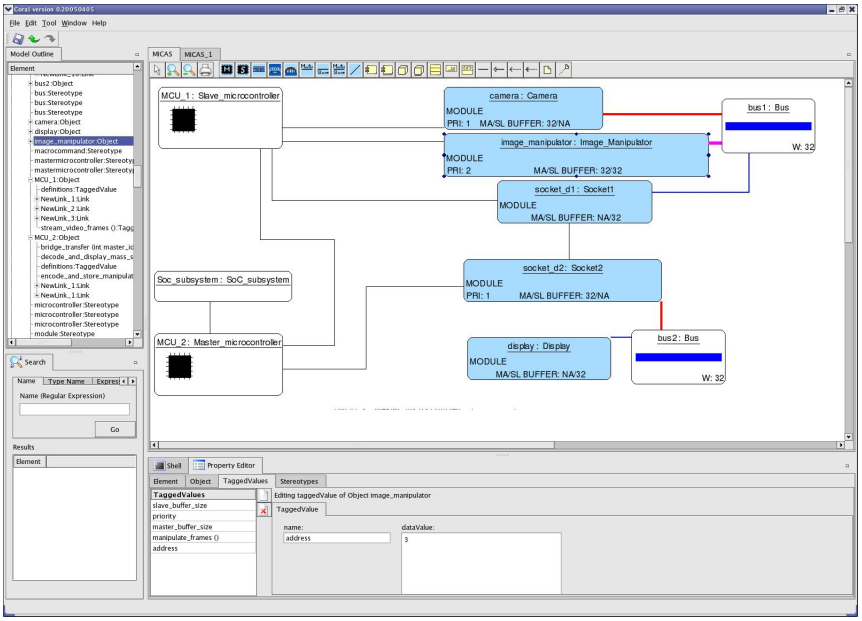


Fig. 3. The canonical example in the MICAS tool.

The application functions as follows. The camera and the image manipulator are in a different domain from the screen. We assume that the screen-domain is the master domain that is controlled by a RTOS (the SoC_subsystem in the diagram). From the RTOS the master domain obtains a *macro-command* `stream-picture` (not shown in the figure). This macro-command is a service provided by the screen-domain MICAS component. We will ignore the discovery of service process needed for the the screen-domain to discover that the camera-domain is connected. The screen-domain now starts allocating *streams*, i.e. connections on the busses. A stream may have parameters like bandwidth etc. The needed stream on the screen-domain is the connection of the socket to the screen. The screen is then put into a listening mode, i.e. when data will be streaming on the bus the screen module will display it. Next the screen-domain *tunnels* a macro-command `stream-picture-to-socket` to the camera-domains micro-controller. The micro-controller on the camera-domain now sets up the needed streams, from the camera to the image-manipulator and the socket, and starts the stream. When the master-domain receives the command *stop-streaming* then it starts the inverse process of tearing down the communication and stopping the modules. A number of abstractions are involved at this level of the design. First the components are described on a high-level of abstraction, quite a lot of structural detail is missing and this will be added in the transformation stage. Also the designer does not specified any kind of bus protocol (streaming, block transfer, etc.). The idea is that at this level of the design the designer will only

specify QoS properties, like required bandwidths. The transformation tool will eventually be used to explore different physical bus implementations.

Designing a MICAS system then consists of 2 tasks:

1. Configuring an instance of the MICAS hardware: i.e. defining what domains exist, what modules exist in a domain, and how they are interconnected with busses.
2. Programming the services of a MICAS domain: i.e. defining what macro-commands are available in which domain.

We call the resulting UML models the MICAS structural model, and the MICAS dynamic model respectively.

From an MBD point of view (c.f. figure 1) the MICAS platform consists of the MICAS hardware components (modules, busses, bridges, sockets) that exist, and the rules that govern the interconnection of these components. The act of designing a specific platform instance (that can be done using the tool described below) then is the act of architecting an application onto the MICAS platform.

3 The MICAS SystemC simulator

This section describes the MICAS SystemC [6] simulator and its constituent components and concepts. The simulator is one possible realization of the MICAS platform. We have made some specific design-choices as to the specific busses that we use, and to the way the modules are interfaced with the micro-controller. Specifically we postulate that all MICAS modules will interface to a bus through an OCP interface [4], and that the all modules will reside in a segment of the memory space of the micro-controller.

OCP is standard proposed for interfacing IP-blocks to busses. OCP is an abstract bus-specification in the sense that it specifies the services and protocols that the bus must satisfy, but it does not specify the physical properties of the bus. In our simulator the physical bus is an AMBA bus [2] a bus available e.g as the physical interface of ARM processors. Each MICAS module provides an OCP interface. To connect it to the AMBA bus it needs to be connected trough an OCP-AMBA converter, that translates between the OCP and AMBA signaling. The SystemC converter modules are parameterized on the bus-width, that can be specified by the designer.

The control of modules in a domain is done by the micro-controller. Each module has a number of control-register that reside in the memory-space of the micro-controller at specific addresses. Sending a command to a module then consists of writing to the memory location of the modules control-register. Sometimes the modules need to communicate with the micro-controller. This is accomplished through an interrupt mechanism. Since the number of MICAS modules in a domain is not constrained and most micro-controllers provide only a maximum of 8 interrupts we have implemented an interrupt controller that is parameterizable in the number of interrupts. At the moment the maximum number of interrupts is 256, which is specified as a constant in the SystemC file. The

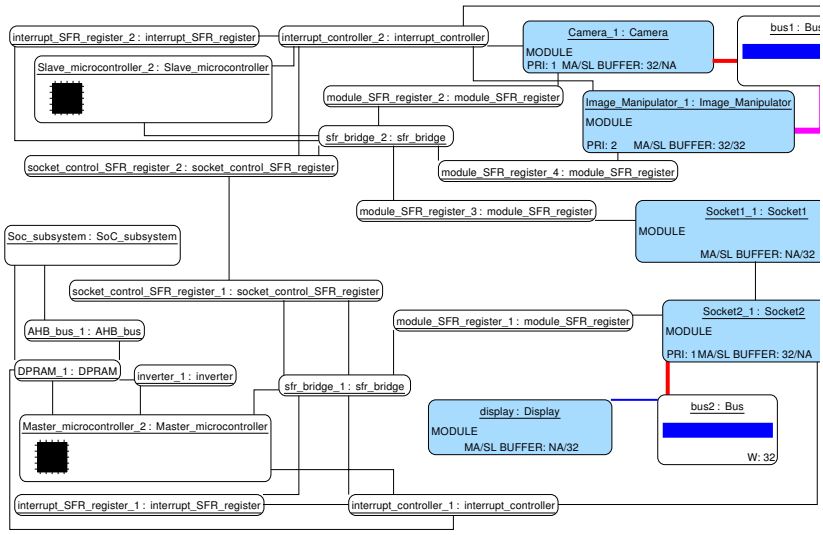


Fig. 4. An example MICAS configuration

micro-controller we have chosen to work with is an asynchronous version of the industry standard 8051 micro-controller [1].

Assume now that we have a MICAS model described at the abstraction level described in the previous section 3. The task of the designer is the to map (i.e. architect) this model onto the components of the SystemC library, adding the bits and pieces that are missing. Figure 4 shows parts of the result of mapping the system onto the SystemC library (the full mapping would be too big to fit the page).

We will start describing the picture starting from the SoC_subsystem to the middle-left. The SoC_subsystem represents the controlling entity for the MICAS master domain. It communicates with the MICAS system trough a dual-ported memory (DPRAM_1), to which it connects through an AMBA bus (AHB_bus_1). The memory communicates with the micro-controller using an interrupt. The data ports of the DPRAM_1 are thus directly connected to the micro-controller, and there is a connection to an interrupt-controller (interrupt_controller_1). The interrupt-controller connects to the micro-controllers interrupt port, and to a interrupt_SFR_register that contains the information about the interrupt number to service. This register is then connected to a sfr_bridge that acts as a synchronous/asynchronous bridge. The sfr_bridge connects to the different control registers of the MICAS modules and sockets (socket_control_SFR_register_1 and modules_SFR_register_1). It should by now be obvious to the reader that this concrete configuration can be derived automatically from the model in Figure 3. Indeed this is the case as we shall see in the next section. A final point to note is that neither in Figure 3 nor in Figure 4 have we specified any type for the connections (the lines), nor have we specified

exactly to which ports in the hardware components the should be connected. It turns out that this information can be deduced from the diagram because it is part of the structural constraints of the MICAS profile.

4 The MICAS Editor

Coral [14] is a highly customizable modeling tool based on the OMG standards. Coral is a metamodel-based tool and it allows the user to define a new modeling language by creating a new model. Coral supports the UML as just one of its metamodels. It is possible to extended UML or to combine it with other user-defined modeling languages. Coral implements the OMG XMI and XMI[DI] standards for model interchange. Therefore, it is possible to interchange models with commercial modeling tools. If the other tool supports XMI[DI], Coral can also exchange diagrams without any loose of information.

We can create new diagram editors to edit models in Coral. This is of course necessary when providing tool support for a new modeling language, but it is also possible to redefine the diagrammatic representation of existing languages, such as UML diagrams. This is useful when we extended the UML language with domain-specific profiles using stereotypes and tagged values. In this case, the appearance of UML diagrams can be redefine based on the presence of these stereotypes and tagged values. However, since the models are based on the standard UML, it is still possible to interchange the models with other modeling tools. Finally, we should note that the Coral tool provides a high-level API to query, transform and generate code from models. By using Coral we have been able to create a tool prototype and use it to validate the approach described in the paper. The Coral modeling tool is open source and it can be download from `http://mde.abo.fi`. The MICAS editor is built as a profile in Coral. A profile makes it possible to extend the basic functionality of Coral by adding specialized features. These features typically include adding actions for context-menus, toolbars and diagram editors, and can be designed to have direct access to the models loaded into the profile and manipulate them at run-time. Within a profile it is also possible to customize the visual appearance of the elements in the diagram editor, making it possible for an element to be rendered differently depending on the underlying model data. Due to the nature of the Coral profile mechanism, the profiles are separated from core Coral, enabling loading and unloading of a profile without losing the core Coral functionality. The MICAS profile for Coral is built upon the standard UML 1.4 profile in Coral. This has been accomplished without modifying the structure of the UML 1.4 language itself – the profile uses a subset of standard UML 1.4 constructs to store all model data. This preserves compatibility with UML 1.4 tools, with the exception of any customized rendering, should the MICAS profile be unavailable. However, using this technique, there must be clear semantics of any element used in the models.

A conceptual overview of the MICAS modeling language is shown in Figure 5. This language is a description on a higher level, the actual constructs used when

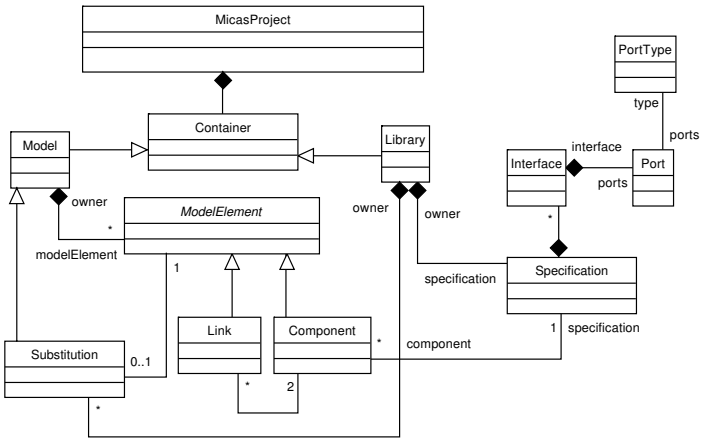


Fig. 5. A conceptual metamodel for the MICAS modeling language

creating the models are expressed using UML 1.4. The MICAS language uses Objects to represent a component in a design, and Class for specifications and ports. Component connections are specified with Links. Classification of MICAS language constructs are implemented using Stereotypes. Parameters needed in the components are specified with TaggedValues. By using a different MICAS profile in Coral, it is possible to redefine the semantics and the visual appearance of the elements and create a modeling language that internally appears as UML, but for the user is a different language.

A MICAS project consists of two parts, the library and the actual model. The first part is a library that is a static model containing specifications for the components used in a MICAS model. The specifications contain information about the interfaces a component has in order to connect to another component in a design. The interfaces in turn specify which ports are available and the type of the port. Since each port has a designated role, the port type serves as classification of the port, providing information about size, direction and its purpose in a design. This information is used to determine exactly to which port in a connected interface in another component a port should connect to. Additionally, a library can contain a set of substitution models that can be used for substituting a component in a design. This step will be discussed further below. The second part of the project is the model containing the actual design. When creating a project, the user has access to one or more libraries that contain components that can be used in the designs. Coral provides the ability to assign toolbar actions to add skeleton components for the component types to the designs. The skeleton components contain a set of default values that later can be modified. These components do not have a specification by default - the user needs to select a specification from the libraries that specifies the role of the component in the design. When the components have been added to the

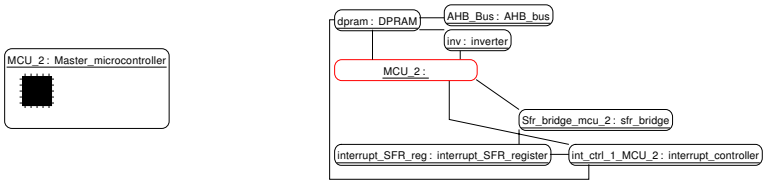


Fig. 6. The substitution model for a master micro-controller.

design, the user can then link the components together specifying component interaction, i.e connecting the interfaces together. Since a component can have several interfaces, Coral provides specialized links for commonly used interfaces. Most notably, the design the user creates with the library, is a simplification of the full configuration. The user design contains only the component needed to determine a specific configuration, additional components needed for e.g. a specific micro-controller can be left out to give the user an opportunity to focus more on the actual layout of configurable components. The transformation step in the work-flow, which is automated, takes care of additional components needed to provide a fully detailed design that can be mapped to executable code.

The transformation is based on a set of rules that the library creator has set up. Every rule provides a matching criteria, a set of *edge relinking rules* and a *substitution model*. The matching criteria in our case is the type of the element and matches only one element at a time. Edge relinking rules are needed to re-connect the edges properly to the substitution model. Figure 6 shows the substitution mode for a master micro-controller. When a match is found it replaces that element with the contents from the substitution model. To avoid loosing the information in the replaced element, the element is copied into the substitution model. After that all the edges that were connected to the replaced element have to be re-linked to the elements in the substitution model. The result of this relinking process can be seen in figure 4, and was described at the bottom of page 8. Edge relinking rules works as follows. For each substitution rule there has to be at least one edge relinking rule. The edge relinking rules are compared to all the adjacent edges of the replaced element. The criteria for choosing the right relinking rule is the type of the connected element. A relinking rule simply tells the transformator where in the substitution model to connect the edges from the replaced element. The transformation always works on a copy of the input model. This way the transformation doesn't destroy any of the initial information. The idea behind the transformation is to add the most basic elements to the model. These elements are often converters from asynchronous to synchronous signals and internal parts of the micro-controller. The elements that the model creator adds to the model are the more complex elements that often need user input. In most cases it is enough to apply transformations on components in the model - however, there are some cases when we also need to transform edges, but this case will not be discussed here.

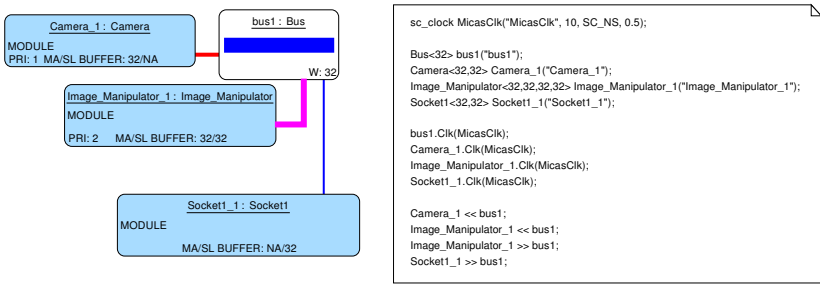


Fig. 7. A example code generation transformation

After running the transformation with all its rules a new model is generated (e.g. the model in figure 4. This model is a closer representation of the hardware design that will be translated into SystemC modules. Now the code generation can read this model from top to bottom and create the SystemC code. It is a straightforward mapping of one model component per SystemC module. The first thing that is generated when you run the code generation is the C++ module declarations. After that all the edges from the model are parsed to determine all the connections between the components. For each edge there will be a function call to an overloaded `operator«` function. Inside this function all the ports that the two component have in common will be connected. By having only one function call per edge the main file will be much cleaner and readable than having all the signals connected in the same place. For each pair of component types that are connected, there will be an overloaded operator function. To know which ports should be connected the ports are parsed from the component specification and passed to a port matching algorithm. This algorithm compares all ports from one component to all ports in the other component and returns the set of connectable port pairs. The actual matching is simply done by comparing the stereotype of one port with the stereotype of the other port. If they have the same stereotype they are considered to match. However, only input ports can be connected to output ports so this is also checked. A concrete example of the result of code generation is shown in figure 7. The figure should be self-evident. For each MICAS component a corresponding SystemC object is created. The connections are set up, and a clock is created.

5 Summary and future work

We have presented a tool that allows us to generate a SystemC simulator for a MICAS application. The starting point of the generation is a model of the application described using the UML MICAS profile. The generation process consists of a number of well defined transformations that add detail to this model until it reaches a level from which there exists an 1-1 mapping onto the concepts of the SystemC library.

A number of things remain to be done. The MICAS simulation framework is not complete yet. We plan to add proper modelling of QoS concepts for the busses, which will also imply that we have to design a routing mechanism. We also plan to add a better programming interface for macro-commands. This will involve the design of an action language with its own profile to properly intergate it into the Coral tool. Currently we are already exploring an extension of our techniques to the mapping of applications to both Symbian and MICAS.

References

1. 80c51 general description and datasheet. `http://www.semiconductors.philips.com/cgi-bin/pldb/pip/p87c554sbbd.html`.
2. Amba overview and specification (rev 2.0). `http://www.arm.com/products/solutions/AMBAOverview.html`.
3. Andromda. `http://www.andromda.org/`.
4. Open core protocol community. `http://www.ocpip.org/home`.
5. Sonics' technical overview. white paper. `http://www.sonicsinc.com/sonics/support/documentation/`.
6. Systemc community. `http://www.systemc.org`.
7. Marcus Alanen, Johan Lilius, Ivan Porres, and Dragos Truscan. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter A Case Study on Developing Protocol Processing Applications Using a Model-Driven Approach. Springer Verlag, 2005.
8. A. Andriahantenaina. Spin: a scalable, packet switched, on-chip micro-network. In *Design Automation and Test in Europe Conference (DATE'2003)*, pages 70–73, 2003.
9. L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *IEEE Computer*, 35(1):70–78, Jan 2002.
10. J. Liang, S. Swaminathan, and R. Tessier. asoc: A scalable, single-chip communications architecture. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 37–46, 2000.
11. E. Mattan. Stream architectures - efficiency and programmablity. In *2004 International Symposium on System-on-Chip*, 2004.
12. A. V. Mello, L. Ost, N. Calazans, and F. Moraes. Evaluation of routing algorithms in mesh based nocs. Technical report, Facultade de Informatica PUCRS - Brazil., 2003.
13. Ian Oliver. Model based testing and refinement in mda based development. In *Forum on Design Languages 2004, Lille, France*, 2004.
14. I. Porres. A toolkit for model manipulation. *Software and Systems Modeling*, 2(4), December 2003.
15. Dragos Truscan, Jõao M. Fernandes, and Johan Lilius. Tool support for uml- dfd model-based transformations. In *1th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2004)*. IEEE Computer Society, 2004.

Tool Support for Quality-Driven Design

Jakub Rudzki[1], Imed Hammouda[2], and Tommi Mikkonen[2]

[1] Solita Oy, Satakunnankatu 18 A, 33210 Tampere, Finland
[2] Tampere University of Technology, P.O.BOX 553, 33101 Tampere, Finland
jakub.rudzki@solita.fi, imed.hammouda@tut.fi, tommi.mikkonen@tut.fi

Abstract. This work presents an example of quality-driven design tool support. The tool investigated is MADE (Modeling and Architecting Design Environment) - a pattern-based modelling tool. In the proposed approach, a set of alternative design solutions are gathered and linked with some quality attributes. As an example, the solutions are applied to a design of remote interfaces with distinct quality requirements, which are high performance and high flexibility. The tool allows to select one of defined solutions and ensures that the design contains all required elements. It is concluded that the MADE tool provides entry level quality-driven tool support and has potential to provide wider quality support with extended solution library.

1 Introduction

Constant quality improvement in software production is essential not only from a technical point of view but also as an assurance of customer satisfaction and business success. Even though the quality itself can be perceived differently from the end user and the development team point of view [17], in most general terms it can be defined as a fulfilment of certain requirements imposed on a final software product [17]. The requirements may concern many aspects of the software, and they can be defined in terms of levels of quality attributes. For example, they can specify performance-related requirements, security, maintainability requirements, etc. While all these quality attributes, defined for a particular software system, determine its behaviour, they also impose or restrict certain design solutions [4,8]. The fulfilment of the required quality attributes is not an easy task, since many of the requirements are dependent on each other and an improvement in one can deteriorate another one [7]. Therefore, it is important not only to know the design strategies to accomplish one quality requirement, but also the potential conflicts between the requirements.

During software production many strategies [4,6,8,20] are used to ensure that the required quality attributes are met. The most common strategy is to verify the requirements fulfilment at a late development stage by studying the whole software system (or a part of it). Such strategy relies heavily on the experience of the design team. Since most of the quality attributes are very closely linked with the system architecture [4,8], late adjustments of once-taken design decisions are costly and difficult to perform. For example, if a system has been designed

as a highly modular distributed system, it may have problems with meeting requirement of high performance due to excessive remote communication. On the other hand, if a system has been designed to be one monolithic piece of software, but it is required to be highly flexible, the requirement may not be easy to fulfil either. In all the above examples, any corrections at a late stage of a design process would prove to be extremely difficult and costly. Therefore, early introduction of quality attributes into the design process increases the chances that the final design will be optimal in terms of quality requirements.

Additionally, in order to design a system in accordance with required quality attributes, the designer must know the most optimal solutions that improve particular quality characteristics. The designer can rely on their experience or can utilise a design tool. Finally, even when an optimal design solution is chosen (based on experience or selected by a design tool), it is essential that the solution is applied correctly. A design tool can help to correctly implement a certain solution also at the implementation stage. Therefore, our research problem focuses on the kind of tool support aiding the design process, implementation, and ensuring that the final software system meets the quality requirements.

In this paper we propose a tool that assists a designer during the design process in creating a design with constant and conscious design solutions that have particular impact on the overall software quality. This tool allows a designer to select proven design strategies (e.g., design patterns) that influence quality attributes. The tool also addresses the problem of a correct implementation of the selected solution by ensuring that essential parts of the solution are implemented.

This paper is structured as follows. In Section 2, quality attributes and related design problems are discussed. Section 3 provides an overview of a tool supporting quality-driven design. In Section 4, the case study used for the tool evaluation is presented. Related work is discussed in Section 5. Finally, conclusions are presented in Section 6.

2 Quality Attributes

Quality attributes can be of very diverse nature. Some quality attributes can be expressed directly in terms of numbers, while others, on the other hand, can be difficult to define quantitatively. For example, performance can be expressed as throughput (in number of operations per unit of time), or as average response time (average time that is needed to complete an operation). Therefore, performance as a quality attribute can be precisely expressed in numbers. However, usability for example, is a very abstract concept that is difficult to define quantitatively. Still, even this quality attribute can be defined. For example, it can be realised as a requirement of highly customisable user interface. In that case, it is still possible to find a suitable design strategy that allows customising the user interface, despite the fact that it is a feature that cannot be directly measured. Other examples of quality attributes include flexibility and modularity. These quality attributes are linked with class or package coupling (dependency) metrics and can be estimated already during design stage.

Based on the above examples, it is possible to identify the following apparent groups of software quality attributes: attributes that are directly measurable, but cannot be measured at the design stage; attributes that are not measurable at all; and attributes that are possible to estimate using a set of metrics already at the design stage. Regardless of the group a quality attribute belongs to, it is possible to identify some design strategies that improve the attribute [2,3,8]. Although the fulfilment of the quality-related requirements cannot be fully evaluated at the design stage (for example performance must be measured) it is possible to use all strategies that encourage best results for certain quality attributes.

The design of a software system brings up a few quality-related problems. First, it is the ability to identify a correct design strategy to address a quality-related problem. The second problem is a correct implementation, in terms of design realisation not only coding, that contains all the essential parts of the chosen solution. Finally, the design choices should be documented in order to give a clear indication of chosen design strategies.

3 Design Tools

3.1 Design Tool Features

A desired tool that supports a quality-driven design process should have certain properties that would allow it to be truly useful. The most important properties should include:

- ability to define desired quality requirements,
- prioritisation of the requirements,
- provision of a number of proven design solutions benefiting quality attributes,
- flexibility for adding new design solutions and quality attribute definitions,
- ability to apply a chosen solution in a controlled manner,
- identification of quality attribute conflicts,
- ability to always allow the designer to have the final decision,
- ability to backtracking design decisions applied to a system.

Even though this list contains many properties that are difficult to implement especially for a fully automatic tool, some of them are possible to address by the tool presented in this work. The biggest benefit of a tool that would possess all the previously-mentioned properties would be a standardised yet flexible way of designing software ensuring that design decisions are taken with quality requirements in mind. This, in turn, would result in a better design and in some cases identification of strong requirement conflicts that are impossible to solve.

The definition of design solutions that support certain quality attributes is a complex problem. This issue can be approached in many ways. For example, based on a set of metrics that can be estimated during the design, well-known solutions, and even experimental data, it is possible to select design strategies that promote particular quality attributes. At the same time, it should be considered how the solutions influence other quality attributes to identify potential

conflicts between quality requirements [7]. Having identified the design solutions, they can be used to provide a number of design suggestions to a designer during the design phase. It would be sufficient if the solutions just indicated a better option, not in quantitative terms. The suggestions would be anyway helpful for a designer. The suggestions could be expressed informally that design *A* is the most optimised for a particular quality attribute *q1* whereas design *B* provides a compromised solution that have a sufficient positive influence on the same attribute *q1* but at the same time the solution does not worsen another quality attribute *q2*. An extensive set of collected design solutions can be built based on measurements or just gained experience, which can be also limited to specific domain.

After obtaining a set of design strategies linked with quality requirements, they would create a core part of the tool. The tool's logic must provide the ability to choose one solution and to ensure that it is correctly applied. The solutions could create a structure of patterns that used one by one during the design process would result in a full design determined by quality requirements.

3.2 MADE Tool

The "Modeling and Architecting Design Environment" (MADE)[3] [13] tool is a design tool that allows creating patterns that correspond to certain UML class diagram structures. An example UML diagram, the corresponding MADE pattern, and a list of tasks are depicted in Figure 1. The figure presents two solutions *A* and *B*, solution *B* has been selected and its elements are presented with striped background. The MADE's elements are referred to as roles (in this case class roles), which correspond to the classes in the UML class diagram. For the selected solution *B*, the abstract class *B* has its representation in the pattern as role *AbstractB*, and the concrete implementation *B* corresponds to the role *BImplementation*. In the context of MADE, the term pattern refers to any configuration of roles, constrains, relations, and dependencies. A MADE pattern can, for example, define what kind of a class should be created - its visibility, attributes, and methods. Constrains can be applied to a method name, for instance, so that the name must have a specific prefix. Also relations to other classes can be defined. A class can be required to extend another class. After defining a set of patterns they can be applied to an implementation. In that case, the applied pattern creates a list of tasks that the implementer has to follow. Some of the tasks are mandatory, whereas others can be optional. MADE ensures that all mandatory tasks are completed, otherwise errors in the design are indicated.

MADE patterns are applied to UML class diagrams. Therefore, patterns can define any model elements that are allowed by UML, not necessarily classes. So far MADE has been used to define complex systems in order to identify and track possible design violations during system maintenance and refactoring [12]. If an update of a MADE-modelled system was performed, any negative effects in other

[3] http://practise.cs.tut.fi/∼mda/

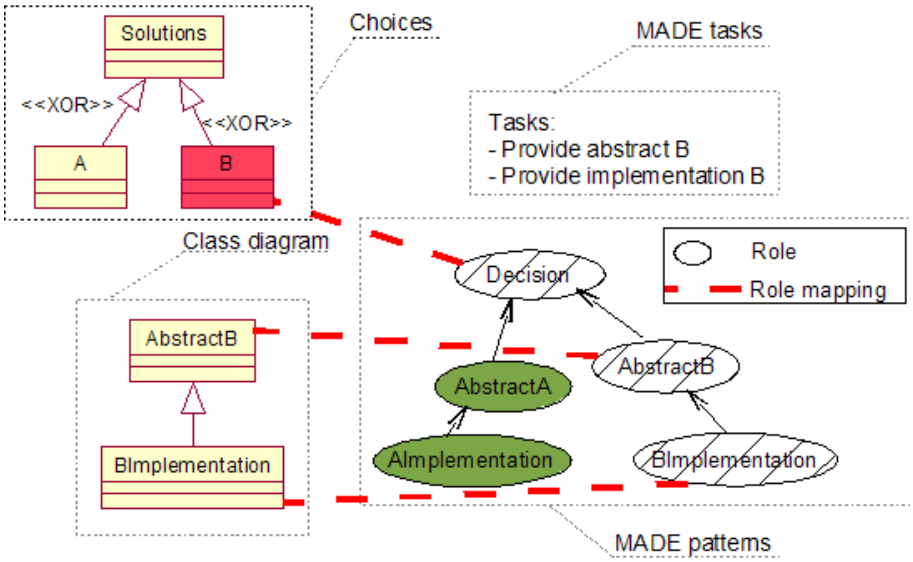


Fig. 1. MADE patterns and UML diagram example

parts of the system would be detected. The ability of the tool to model complex UML structures as systems of patterns qualify it as an example tool that can support quality-driven design. Since it is possible to define patterns that apply some constrains on a UML class diagram, a set of patterns can represent different design solutions. The solutions for one problem should represent alternative designs that differ only by some influence on specific quality attributes.

Having a set of design solutions and defined quality requirements, a designer can select a solution that is the most suitable for a particular problem. In that case, a pattern is applicable only to a part of a system, so that in each step a designer can choose the best solution. The choice does not have to be dictated by the same quality attribute in all cases, but the tool can give alternatives and indicate their influence on certain quality attributes.

4 Case Study

To verify the applicability of the quality-driven tool support we used the MADE environment as an example design tool to solve a sample design problem.

4.1 Design Problem

In order to demonstrate the benefits of a quality-driven design tool support, the MADE tool was used to design a simple system. The example system was a sim-

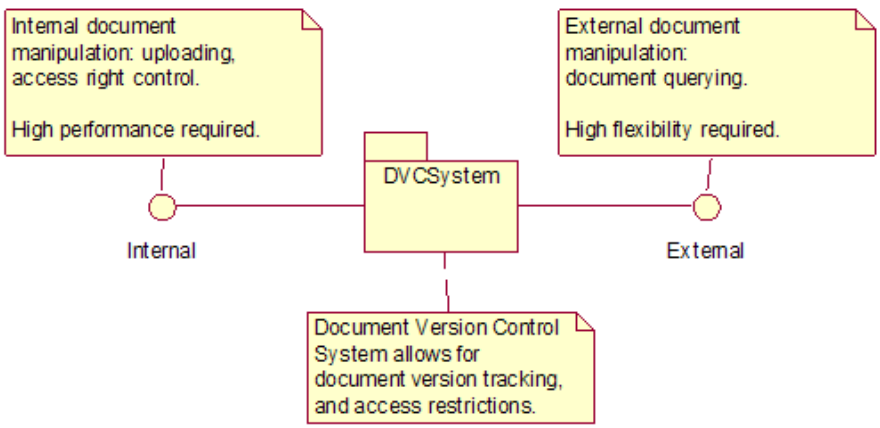


Fig. 2. Document Version Control system architecture overview

ple Document Version Control system[4]. A general architecture of this system is depicted in Figure 2. The system was meant to be remotely accessible. The quality requirements for the system were high performance for all internal operations (document uploading, access right management, etc.) and high flexibility for the external client operations (mostly document querying). As a consequence, two interfaces were to be designed. One internal interface that provided the highest level of performance and external interface that ensured high flexibility.

4.2 Solution

In order to solve this design problem using the MADE tool, a set of alternative design solutions had been defined in terms of patterns that later could be used to generate a real implementation. Since the problem of performance in distributed systems has been discussed a number of times, for example, concerning the J2EE technology [9,1,22], and additionally some empirical results have been presented [24], it was possible to define three example alternative design solutions for remote interfaces. The solutions were based on two well-known design patterns Facade and Command [10]. Two solutions were direct implementations of the design patterns, however, the third solution Combined Command was based on the Command pattern, but provided a possibility to reduce the number of remote calls required to complete a task. All three design alternatives are presented in Figure 3.

In general, design patterns are used numerous times [1,22] in distributed systems; naturally their implementations must be adopted to be suitable in that domain. In the case of J2EE, for example, the Facade can be implemented as

[4] The functionality of the system itself is not relevant for the demonstration

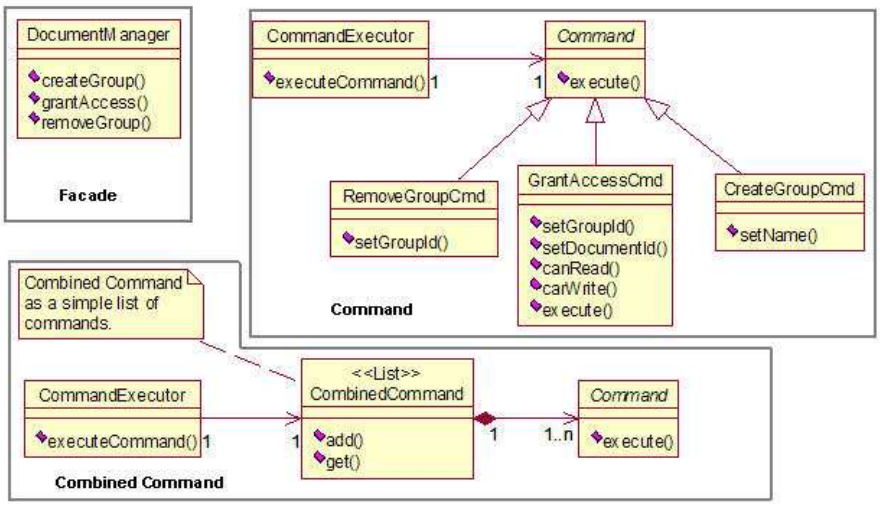


Fig. 3. Alternative design solutions of a remote interface

a Session Bean that exposes service methods to the clients. The Command design pattern, however, is usually implemented as a set of Command classes that take parameters as attributes and a method that executes the command. In that Command design pattern implementation a client creates and initiates appropriate command objects with required parameters, then the commands are passed to the remote location (server-side) and executed. Finally, a command object is returned to the client and then all execution results can be retrieved from the command object. In the case of Combined Command, the design pattern can be implemented as a simple list with command objects that are passed and executed at a remote location. The only difference comparing to a single command is that a number of commands are executed as a result of only one method call. In any variant of the Command solution, the solution requires an abstract Command class with common functionality needed for command execution, a set of Command classes with specific functionality required by the client, and a command executor on the remote location (server-side).

After the design solutions had been defined, they had to be linked with quality attributes that later could be used for taking optimal design decisions. Based on experimental data [24] the Facade solution was defined as a high performance design for remote interfaces, but with a fixed interface. The Command solution, on the other hand, was degraded in terms of performance, but its interface was flexible. The Combined Command solution was a compromised solution that potentially could achieve better performance than Command and Facade, but only if method call reduction was possible.

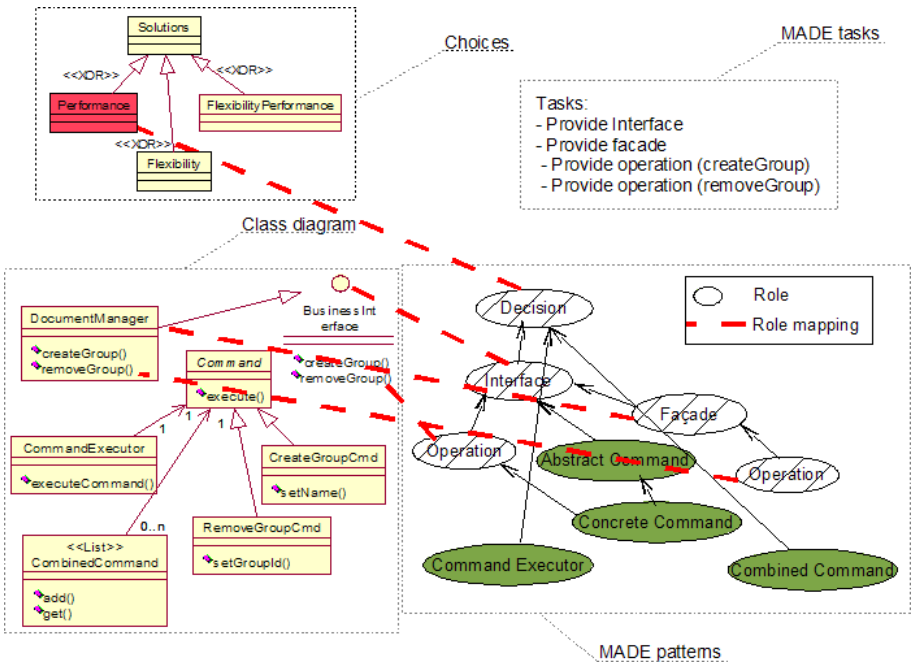


Fig. 4. MADE patterns and UML diagram for performance solution

4.3 Implementing Solutions in MADE

The design solution roles were alternative options that had to be chosen by a designer. Each solution contained its description with listed benefits and drawbacks on performance and flexibility. The design choices are depicted in Figures 4, 5, and 6 as *Choices*. In order to model the solutions in MADE a starting point defining required methods was needed. Therefore, for each solution a business interface was defined. The interface contained all methods (services) that a particular solution had to provide. As the interface could define any services there were no limits on the number of methods, however, since they had to be accessible for clients they had to be public.

Facade-based Solution. In the case of Facade solution the relation between the business interface and Facade implementation was straightforward in the simplest case. Figure 4 presents facade solution with highlighted (striped background) roles that constitute the solution and mappings to UML classes. The Facade implemented the business interface, so all methods and their signatures remained unchanged. Therefore, in addition to mandatory role for the business interface, the solution include facade implementation role and operation role. A list of tasks provides information to the designer how a design should be generated for the Facade solution.

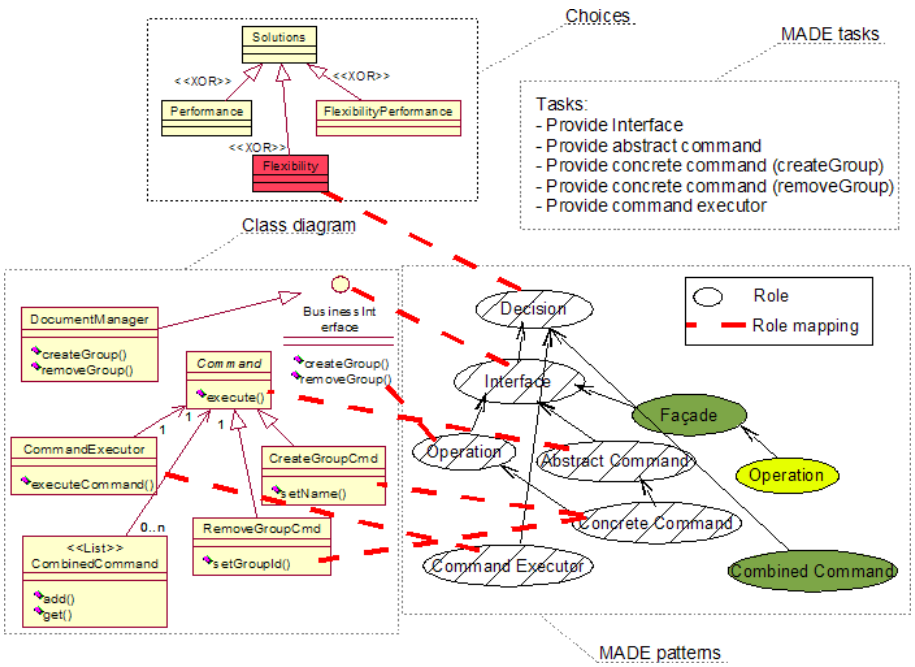


Fig. 5. MADE patterns and UML diagram for flexibility solution

Command-based Solution. The Command solution, depicted in Figure 5, could not be resolved by simple implementation of the business interface. Hence, each service method corresponded to a Command class, and each parameter of the service method was mapped to an attribute of the command. This mapping was defined in MADE as a set of roles: abstract command role, concrete command role, and command executor role. The concrete command role depended on a service method of the business interface. Each service method in the business interface required one concrete command implementation.

Combined Command-based Solution. The Combined Command solution, presented in Figure 6, was practically identical with the Command solution. The only difference between the two solution is that for Command Combined a combined command had to be designed.

The defined design solutions were applied to solve the two design problems related to internal and external interfaces of the example (Document Version Control) system. The MADE design view and corresponding UML diagram in Rational Rose with the defined solutions are shown in Figure 7. As the first step, an internal business interface was defined, the interface contained methods allowing for document creation, update, and access right modification. Next, a solution that provided the highest performance was applied. Based on the provided in design solution descriptions the Facade solution was the most optimal.

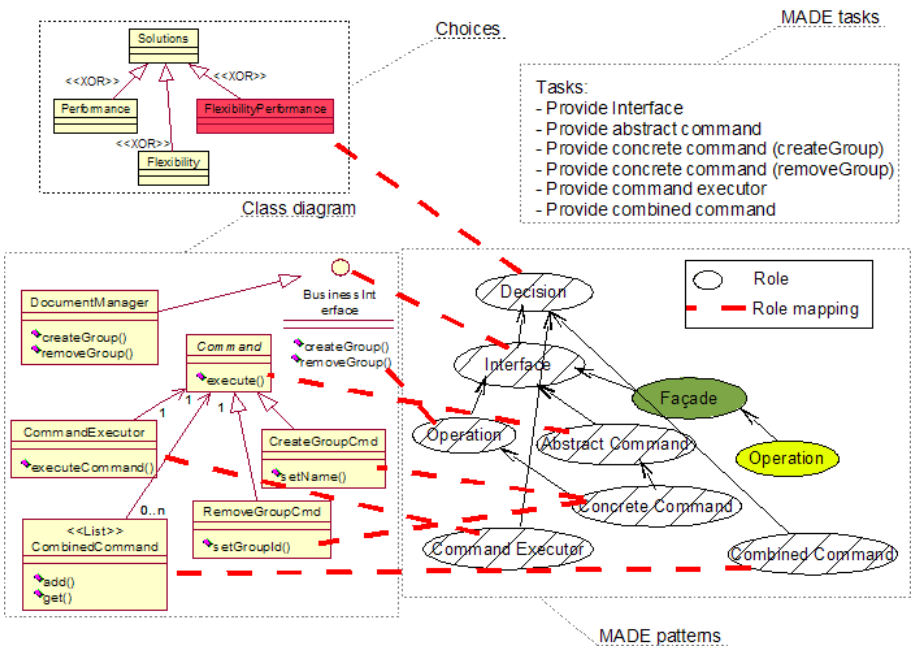
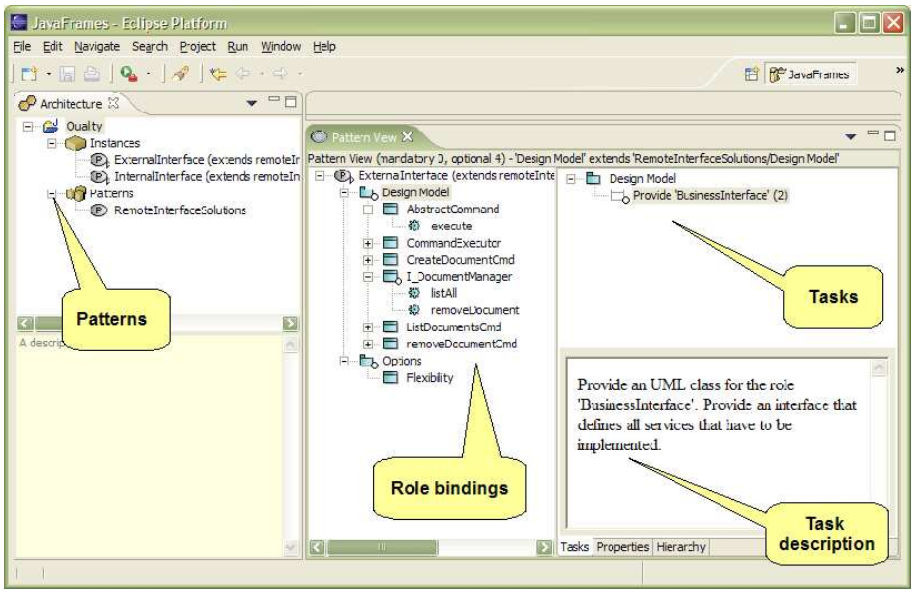


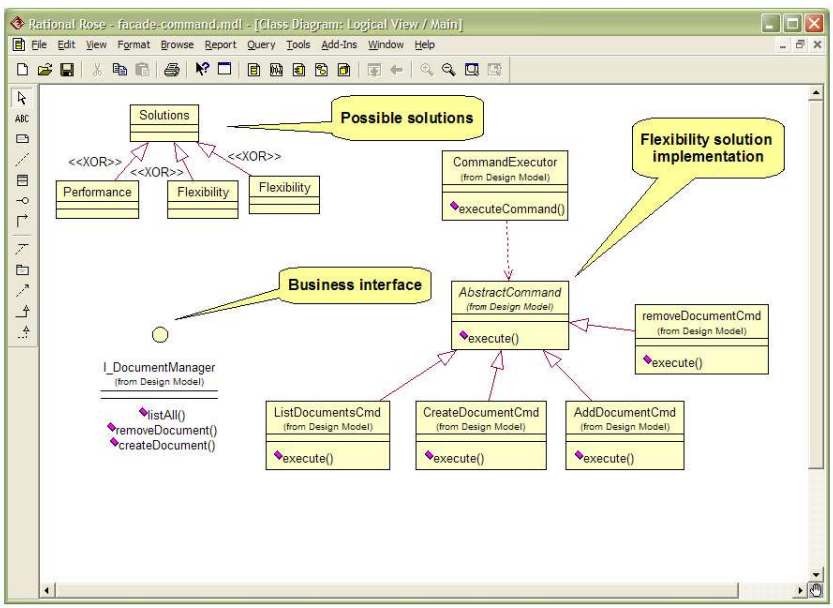
Fig. 6. MADE patterns and UML diagram for compromised solution

After choosing the solutions, a list of tasks was created that define this solution. In this case the list included the following main points: business interface implementation, and implementation of all methods defined in the business interface. If any of the mandatory tasks was not resolved, the task was highlighted and an error was indicated in the task list, as well in the design as a red dot next to the role name. This verifying mechanism ensured that the chosen solution was correctly implemented. The external interface, which was required to provide high flexibility, was designed in a similar way. First, the business interface with methods for document querying was defined. Next, an optimal solution was selected; in this case Command solution was the optimal one. After the solution selection, a list of tasks was generated by MADE. The tasks included: creation of command executor implementation (a helper class where the command classes were executed), creation of abstract command, and finally command classes that corresponded to all methods of the business interface. The order of implementation was important, the abstract command had to be defined before any command class. The order was a logical consequence of the fact that all commands extend the abstract command. As in the Facade case, tasks were marked as unresolved until all tasks were completed.

A design of two interfaces was created as a result of all the above-described steps. One internal interface was designed for high performance, and the other external interface was designed to be highly flexible. The final design has been



(a) MADE design view



(b) UML class diagram in Rational Rose

Fig. 7. Remote interface implementation

based on two design choices dictated by defined quality requirements. Trusting that the solutions provided the most optimal designs in terms of the quality requirements, the final design was the optimal design considering desired quality characteristics.

4.4 Evaluation

The example design showed that MADE tool was able to provide support for a static design (UML class diagram) based on certain known design solutions that could be applied to ensure desired quality levels. The presented case study demonstrated how the tool could be utilised to solve a real design problem. The benefits from using MADE include:

- assurance that, once chosen, a design solution will be implemented correctly (designer must follow a list of tasks).
- Any modifications of already existing design that compromise original design solution are identified and shown (i.e., presented as unresolved bindings).
- Additionally, the design solutions used can be highlighted on a UML diagram, which helps to localise the design parts responsible for particular quality attributes.
- Finally, the MADE tool helps to document design decisions made by showing alternative solutions that have not been used for that system.

The tool allows for much flexibility in using it; the designers are able to create and modify existing design solutions to adjust them to particular needs, for example, to some specific domains.

Based on the presented case, the MADE modelling environment can be considered as a tool that fulfils most of the requirements imposed on a quality-driven support tool, specified in Section 3.1. The are naturally possible improvements in the tool that could include more extensive set of provided design solutions corresponding to quality attributes. Moreover, it would be desirable to include sequence diagrams as additional diagrams used for quality analysis.

5 Related Work

The concept of a tool that would support a software development process and allow monitoring of the software quality is not new. There are already tools that provide support for quality attributes fulfilment during a development process. For example, the 'Metric-Driven Analysis and Feedback' system [25] constantly monitors a set of metrics that correspond to certain quality attributes. The system gives a comprehensive view on the software state and levels of particular metrics that can be adjusted to the project needs. The metric monitoring is beneficial if the software system is already designed correctly, meaning in a way that does not prevent achieving the assumed quality levels. Another approach by Janakiram and Rajasree suggests quality estimation already at the analysis stage [16]. In that solution the quality attributes are taken into account in a very early

stage of software production. Both of the approaches may be complementary solutions to the tool presented in this work that together could increase the overall software quality by constant supervision at all stages of software development.

The ability to measure, or sometimes just to estimate, software quality naturally is related to design decisions, which in turn have impact on quality attributes. As identification of links between certain quality attributes and design decisions is very important, it has been discussed in many publications [2,3,5,4]. The authors present various aspects of design related to quality attributes, e.g., architectural patterns (called attribute primitives) and their impact on specific quality attributes. Moreover impact on additional quality attributes that are not of main interest in particular solution is discussed. Apart for only quality attributes the problem of quality-driven design has been addressed [21,18]. Suggested solutions include subjective design quality evaluation as well as tool-based evaluation [21]. There are also examples of quality analysis of real systems using architectural patterns (quality attribute design primitives) [18], the analysis provide additional information of real systems that can be used to create domain-specific design solutions.

It is also worth mentioning that some of the quality aspects of software can be modelled already at design level. For instance, performance of a system can be modelled as an UML diagram [11] containing artefacts influencing performance. The performance estimation is created based on empirical data, simulations, or assumed values. All the research in this area allows for gathering existing design solutions, as well as defining new ones, that have specific impact on software quality, and utilising them as a base for quality-driven design tool support.

The presented example tool-support for quality-driven design relies heavily on an assumption that the provided design solutions reflect optimal approaches for addressing certain quality requirements. Therefore, a reliable estimation of quality attributes for a particular design solution is vital. The quality attributes can be defined in terms of software metrics. Especially object-oriented metrics have been defined many times [15,19,17]. Moreover, the links between the metrics and the quality of software are presented in many papers [14,23]. For example, metrics for object-oriented design (MOOD) [14] define a set of metrics that are validated theoretically and empirically, so that they provide a reliable view on software system quality.

6 Conclusions

In this paper we introduced an approach for tool-assisted quality-driven design. We used the approach to solve a design problem of creating a remote interface with high performance as the main quality requirement. The tool provides three alternative solutions that have different implications on performance. In the proposed approach, if the quality requirements are conflicting (e.g., performance and flexibility), the designer can manually choose a strategy that is the best in a given situation depending on the strength of certain requirements and their priorities. This approach allows creating designs that were optimised for specific

set of quality attributes. The most important benefit of this approach is that, already at the design stage, the quality attributes are taken into account, which in turn makes more likely that the resulting design and consequently a whole software system will fulfil all quality requirements.

The quality-driven tool support discussed in this paper makes the design process more quality-oriented, and the presented MADE environment is an example tool that fulfils most of requirements for such a tool. Future work should focus on further tool improvement. The improvements might include the ability to analyse an existing design and point design decisions that affect certain quality attributes. Moreover, a language (e.g., quality attribute-specific role types) allowing the definition of quality attributes and requirement conflict would additionally enrich the approach.

7 Acknowledgements

We would like to thank all participants of the 'Software Engineering Research' seminar, organised at Tampere University of Technology and led by Kai Koskimies, for their comments and feedback that helped to improve this paper.

References

1. Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2001.
2. Felix Bachmann and Len Bass. Introduction to the attribute driven design method. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 745–746, Washington, DC, USA, 2001. IEEE Computer Society.
3. Felix Bachmann, Len Bass, and Mark Klein. Moving from quality attribute requirements to architectural decisions. In *STRAW'03 : Second International SofTware Requirements to Architectures Workshop located at ICSE'03*, pages 122–130, Portland, OR, USA, 2003.
4. Len Bass, Mark Klein, and Felix Bachmann. Quality attribute design primitives. Technical Note CMU/SEI-2000-TN-017, Software Engineering Institute (SEI), 2000.
5. Leonard J. Bass, Mark Klein, and Felix Bachmann. Quality attribute design primitives and the attribute driven design method. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 169–186, London, UK, 2002. Springer-Verlag.
6. Jørgen Bøegh, Stefano Depanfilis, Barbara Kitchenham, and Alberto Pasquini. A method for software quality planning, control, and evaluation. *IEEE Softw.*, 16(2):69–77, 1999.
7. Barry W. Boehm and Hoh In. Identifying quality-requirement conflicts. In *ICRE*, page 218, 1996.
8. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

9. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
11. Object Management Group. UML profile for schedulability, performance, and time, version 1.1. Technical Report formal/05-01-02, Object Management Group, Inc., 2005.
12. Imed Hammouda. A tool infrastructure for model-driven development using aspectual patterns. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.
13. Imed Hammouda, Johannes Koskinen, Mika Pussinen, Mika Katara, and Tommi Mikkonen. Adaptable concern-based framework specialization in UML. In *ASE*, pages 78–87, 2004.
14. Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. Softw. Eng.*, 24(6):491–496, 1998.
15. Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
16. D. Janakiram and M. S. Rajasree. Request: Requirements-driven quality estimator. *SIGSOFT Software Engineering Notes*, 30(1):4, 2005.
17. Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
18. Anna Liu, Len Bass, and Mark Klein. Analyzing enterprise javabeans systems using quality attribute design primitives. Technical Note CMU/SEI-2001-TN-025, Software Engineering Institute (SEI), 2001.
19. Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
20. Francisca Losavio. Quality models to design software architecture. *Journal of Object Technology*, 1(4):165–178, September-October 2002.
21. Mika Mäntylä. Developing new approaches for software design quality improvement based on subjective evaluations. In *ICSE*, pages 48–50. IEEE Computer Society, 2004.
22. Floyd Marinescu. *EJB Design Patterns*. The MiddleWare Company, 2002.
23. John C. Munson and Tagi M. Khoshgoftaar. The use of software complexity metrics in software reliability modeling. *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 2–11, 1991.
24. Jakub Rudzki. How design patterns affect application performance - a case of a multi-tier j2ee application. In Nicolas Guelfi, Gianna Reggio, and Alexander B. Romanovsky, editors, *FIDJI*, volume 3409 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 2004.
25. Richard W. Selby, Adam A. Porter, Douglas C. Schmidt, and Jim Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *ICSE*, pages 288–298, 1991.

Model Driven Engineering in Automatic Test Generation

Endre Domiczi[1], Jüri Vain[2]

[1] Mentorite Oy, Akadeemia tee 21, 12618 Tallinn, Estonia
edo@iki.fi
[2] Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia
vain@ioc.ee

Abstract. The work in this paper describes test code generation from a UML model of the System Under Test (SUT) to TTCN-3. The UML model is marked up according to the UML 2 Testing Profile (U2TP) then transformed to a UPPAAL model. UPPAAL is a tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. From the UPPAAL model we generate TTCN-3 code skeletons. New elements in the transformations are the usage of state diagrams and object diagrams (Test Data Pools) as starting points for the transformations. Test generation based on state space exploration frequently generates inefficient tests. We exploit the UPPAAL model-checker features that can check invariant and reachability properties. Test traces skeletons are generated using *environment automata synthesis*. Test goals and strategies are carried over into the UPPAAL environment as temporal and timing constraints. The generated traces are provably correct with respect to these properties.

1 Introduction

The complexity of modern systems has increased significantly and the necessity for thorough and systematic testing is a must. Manual test generation is error-prone, and slow. To make test generation effective it must be automated (often called Computer Aided Test Generation, i.e.CATG) and derived from the original system specification. This also ensures repeatability. Test generation based on the exploration of the state space exploration is helpful but is often inefficient; the tests do not to cover specific parts due to an incomplete or missing specification or at the other extreme it is impossible to execute all the test cases due to the formidable number.

Plain UML models focus primarily on the definition of system structure and behavior. As is, it provides only limited means for describing test objectives and test procedures. The need for solid testing resulted in a UML profile for the testing domain, the UML 2.0 Testing Profile (U2TP) [3,4].

U2TP bridges the gap between design and test by providing means for using UML for both system modeling and test specification. This allows reuse of UML design documents and models for testing and computer-aided test generation .

The UML Testing Profile provides support for the specification of test suites, including test behavior and the generation of a verdict on the correctness of the test case.

Using elements of the U2TP, e.g. stereotypes, the original model is “marked up”. Later on these markings guide the mapping from the original model to test specification model. This is analogous to the approach applied when transferring from PIM to PSM then executable model.

The profile introduces four logical concept groups covering the following aspects: test architecture, test behavior, test data and time

In its present state our test generation methodology and the prototype tool focus on test behavior and data. We assume that the test architecture is defined by some already existing tool such as TT Workbench. Time aspects will be introduced in the methodology as the result of ongoing research.

As a case study, the methodology will be evaluated by applying it to a UML model of the OBSAI (Open Base Station Architecture Initiative) model.

The rest of the paper is structured as follows: After a short introduction of UML Testing Profile in the next section mandatory and optional test aspects of UML Testing Profile are discussed.

Then the transformations related to automatic test generation are described In section 4, we will show a UML model for the software management part of a BTS as described in OBSAI. Section 5 evaluates our testing methodology by applying it to the UML model. Some conclusions are drawn and future work discussed in Conclusion.

2 From PIM to PSM to code

We followed the path recommended by MDA and shown as applied by PathMate in Fig. 1.

The three parts of the toolset cooperate to turn models into executable systems:

• Transformation Maps – Generate code. C, C++, or Java software with off-the-shelf Transformation Maps (or custom maps for other languages or specific platforms)

• Transformation Engine – The Engine transforms platform independent models into working applications using the platform specific markups

• Spotlight – For verify and debugginof the application logic

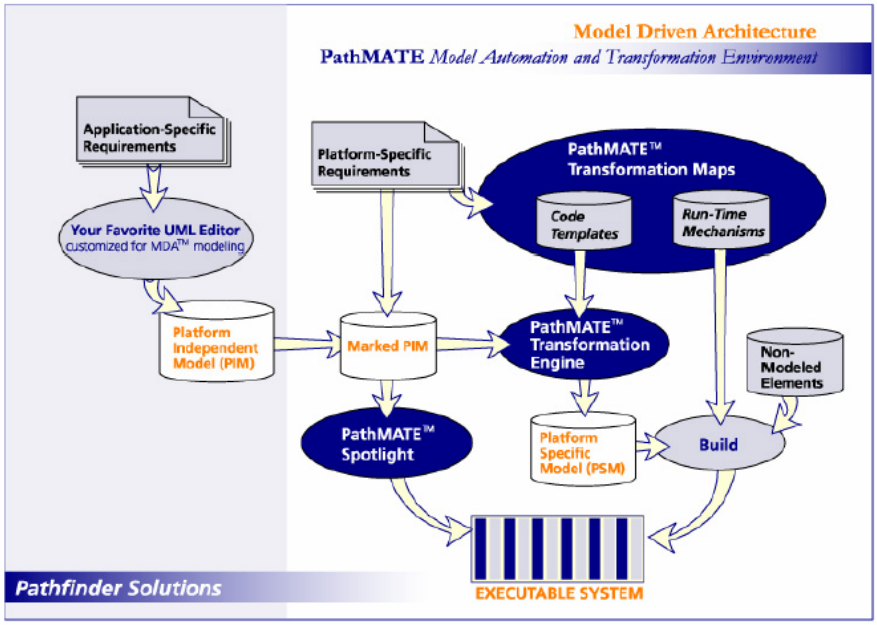


Fig. 1. A toolset for MDA [9]

3 A Case Study: OBSAI

The approach introduced in this paper is illustrated by OBSAI case-study. The OBSAI (Open Base Station Architecture Initiative) family of specifications provides the architecture, function descriptions and minimum requirements for integration of a set of common modules into a base transceiver station (BTS). The example presented here is based on the Software Management (SwM) part of the OAM&P interface at RP1 (Reference Point 1 interface for the Control Plane (C-Plane, a.k.a. "signaling") and Management Plane (MPlane, a.k.a. "OAM&P").) (see Fig. 2.).

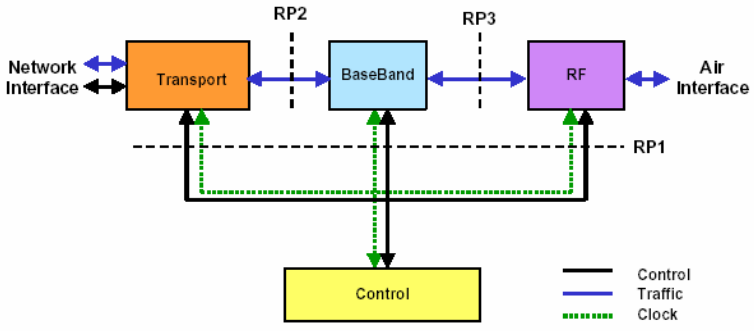


Fig. 2. BTS Reference Architecture The OBSAI operation request scenario corresponds to the "S3 Request/Response" pattern described in [12] (Fig)

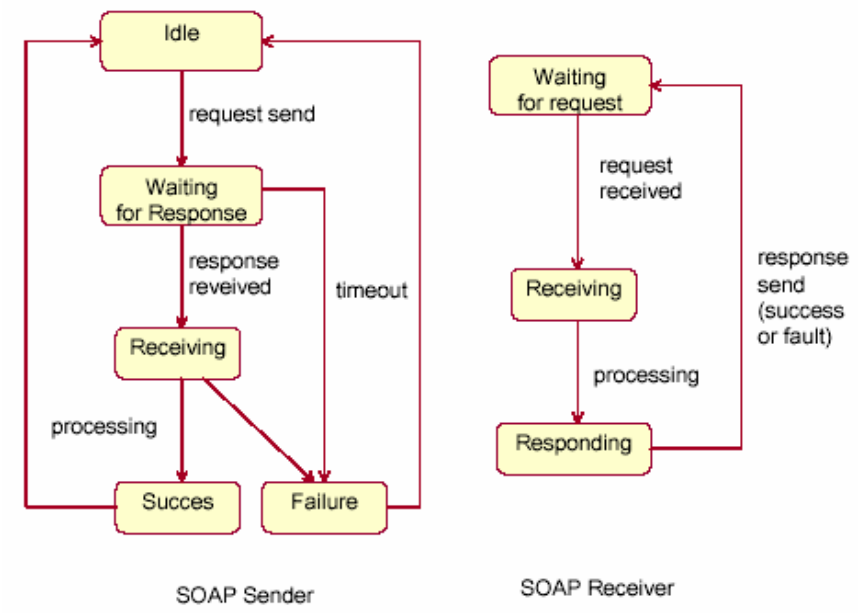


Fig. 3. Operation Request State Model

4 A Methodology for Testing

4.1 Test generation by model transformations

Simon Pickin et al in [2] present a method for the automated synthesis of test cases (with built in oracle in the form of test verdicts) from test objectives described as high-level scenarios. The method was supported by a prototype tool.

The inputs to the method are:

- a set of test objectives, in a UML sequence diagram based notation,
- a UML model of the application, comprising at least a class diagram and a state diagram for each of the main classes,
- a description of the initial state of the application in the form of a UML object or deployment diagram.

Test cases – are represented in a UML sequence diagram based notation – exactly defining the ordering of call sequences and associated test verdicts.

The tools applied were:

- UMLAUT: which manipulates the UML meta-model, enabling automatic model transformation
- TGV is a test synthesis tool based on an on-the-fly and partial traversal of the enumerated state graph of the specification.

We transform UML models using U2TP as markup for the purpose of model transformation with the goal of test generation. In our first "manufactural" version the U2TP markup guides encoding the transformations as Python scripts. Having tested these transformations on our case study application(s) in the next version we wish to use code generators. Candidates are the built-in code generators of the UML tools used for our case study model (Poseidon/Velocity/, Enterprise Architect) or Cheetah a Python-powered template engine and code generator. This latter has the advantage of being vendor independent, "source forge" and gives users the flexibility and power of the Python language. For example Cheetah lets one inherit and extend one template from another.

The set of transformations needed for test generation in our experimental system is following. We assume that as a starting point the SUT (System Under Test) is given in form of UML 2 model with U2TP markup, then the syntax of UML state diagram is transformed to Mealy machine format that carries more restricted semantics. Further the model of SUT is converted to timed automata format of UPPAAL model checker and is extended with its dual so called "environment" automaton and with "observer" automaton that detect the traces relevant to the test goal. The test trace skeletons generated by Uppaal are further extended with concrete test data. From those the TTCN-3 testcase scripts are finally generated. Since for conversions between UML and Uppaal models the XSLT transformation technology and standard parsing are used we focus in the following more on semantic aspects of model transformations.

4.2 A step-by-step description of the transformations

1. Create model in your UML tool.

A fragment of OBSAI case study model is depicted in Fig. 4 as a UML Statechart.

2. Export UML model from your UML tool in XMI format
3. Convert the XMI format UML model into UPPAAL format implemented in Python language.

UPPAAL [13] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

For generating the control skeletons of test traces by Uppaal model checker the test data defined as UML object diagrams are abstracted away using predicate abstraction [10]. It means that input/output symbols of the Mealy automata extracted from SUT UML model denote equivalence classes of the original test data sets. That is the first reduction step of the model state space to be explored by model checking.

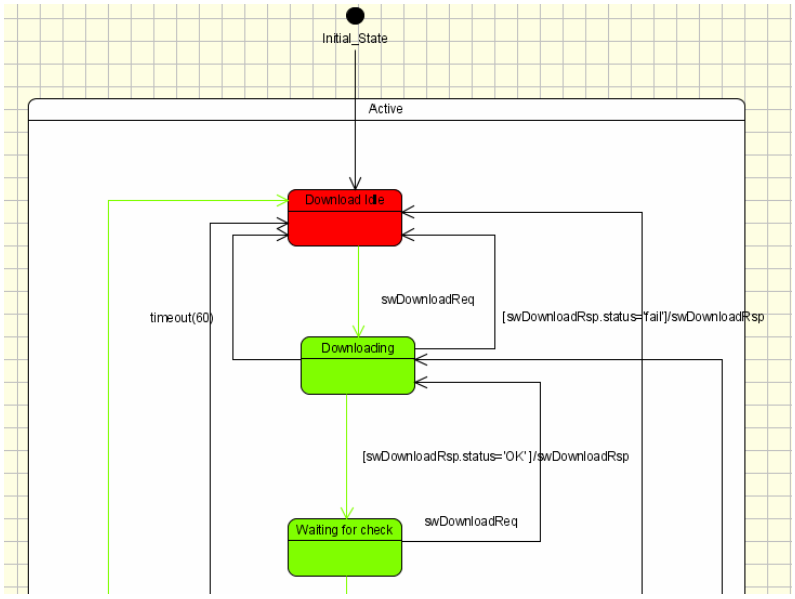


Fig. 4. Statechart in the primary (UML) model

4. Convert Prolog representation of the SUT into UPPAAL and add the test goals to the Prolog representation.

Our target is to synthesize the provably correct (relatively complete and sound) set of test cases regarding the test goal provided. Soundness means that the parallel composition of models used for finding diagnostic traces by model checking procedure has to be able to generate a (potentially infinite) set of traces satisfying the test goal. For obvious reasons we restrict the test generator with goals being satisfied only with a finite set of traces.

Such test goals (see Fig. 5.) are related to the state space exploration, for example:

- All transitions: Every specified transition is exercised once. This exercises all states, all events, and all actions. No particular sequence is required, and any sequence that exercises each transition once will suffice.
- All n-transition sequences: Every specified transition sequence of n events is exercised once.
- All round-trip paths: Every sequence of specified transitions beginning and ending in the same state is exercised once. The shortest round-trip path is a transition that loops back on the same state. A test suite that achieves all round-trip-path coverage will reveal all incorrect or missing event/action pairs.

Relative completeness means that the test goals are restricted to those providing only finite trace sets. It must be stressed that in many cases even finite trace sets are practically undecidable. Therefore we allow practical resource (time and memory) bounds to be specified for our test generator.

Trace generation bases on “closed world” model consisting in addition to SUT automaton also in its dual so called “environment” automaton. The “environment” automaton synthesized using an approach similar to that described in [8] guarantees that all possible input sequences are presented to the SUT automaton. The third component in the model is “observer” automaton that recognizes the SUT automaton

transition sequences satisfying the test goal and signals about that by reaching a pre-defined state “target”. That way the model checking is bound to solving the state reachability problem stated in CTL [11] as `E<> observer.target`.

It must be noted that because of the model construction principles described above the query given is the same for any test generation task regardless of the goal applied.

5. Extracting test data type information from XMI; i.e. the original UML model into TTCN-3
6. Convert the UPPAAL trace into TTCN-3.

The last step in the sequence of transformations refines the abstract test data presented in the trace and selects the proper data item from the test data pool. The TTCN-3 code generator extracts the test data tpe information from XMI of the original SUT UML model and using predefined code templates generates an executable script (Fig. 6).

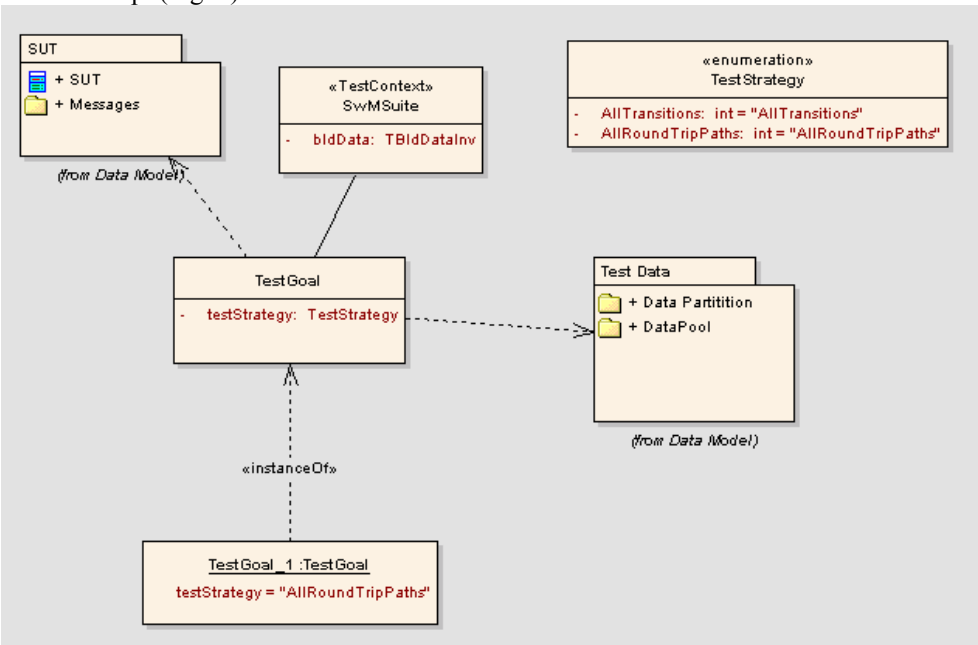


Fig.5. Test goals represented in UML with U2TP stereotypes present

```
module SwM_TC1
{type charstring TFailure ("fileNotAvailable",
"fileIntegrityNOK");
   type record swCheckReq {
      universal charstring filename,
      integer expectedChecksum};
  type port TypePort_1 message {
      inout all};
  type component TypeComponent_1 {
      port TypePort_1 Port_1}
  testcase UpdateSoftware ()
      runs on TypeComponent_1
  {var swDownloadReq swDownloadReq_1;
```

```
swDownloadReq_1.sourceFilename := correctSourceFileSpec";
      swDownloadReq_1.destFilename := "DF1";
      Port_1.send(swDownloadReq_1);
      timer timer1 := 60.0;
      timer1.start;
      template swDownloadResp swDownloadResp_1 :=
      {status := "OK"}
      alt {
        [] Port_1.receive(swDownloadResp_1){}
        [] Port_1.receive {
              setverdict(fail);
              goto TC_END;}
        [] timer1.timeout {
              setverdict(fail);
              goto TC_END;}}
      template swCheckResp swCheckResp_1 :=
      {fileIntegrity := "OK}
      alt {
        [] Port_1.receive(swCheckResp_1) {}
        [] Port_1.receive {
              setverdict(fail);
              goto TC_END;}}
      template swActivateResp swActivateResp_1 :=
      {status := "OK",
            failureReason := ?}
      alt {
        [] Port_1.receive(swActivateResp_1) {}
        [] Port_1.receive {
              setverdict(fail);
              goto TC_END;}}
      setverdict(pass);
      label TC_END;}
  control {execute (UpdateSoftware());}}
  with { encode "SOAP" }
```

Fig. 6. TTCN-3 code

5 New elements in our approach

- We use UML state diagrams and object diagrams as starting points for our set of transformations (test generation).
- We use ENVironment automata synthesis for generation of test traces (skeletons). Test Goals/Objectives are of style “all transactions at least once with the length of traces minimized” or “all traces of length n”, “ percentage n of all transactions covered”
- Temporal and timing constraints are satisfied when generating traces with the UPPAAL engine. The traces are provably correct with respect to properties given

in the test goals/strategies. UPPAAL has been chosen over SPIN, because timing will be used in the future.

6 Conclusion and Outlook

We have been able to generate TTCN-3 scripts starting from the UML model. It was possible to specify test partitions, test data despite lack of proper object diagram support in UML tools.

In the future one should be able to check consistency of the object diagrams with the original UML model. In our specific application the messages had to be simplified because modeling data structures of messages in the tools was not sufficient.

Test goals and strategies were also specified, however not in the "primary" model, i.e. the UML model. A "standardized" way of specifying test goals, e.g. a certain state should be traversed n times, has to be established. This should also be changed in the next version.

Also, code generation templates should be taken into use.

References

1. Michael Ebner: TTCN-3 Test Case Generation from Message Sequence Charts (2004)
2. Simon Pickin, Claude Jard, Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, Alain Le Guennec: System Test Synthesis from UML Models of Distributed Software (2002)
3. UML Testing Profile - Request For Proposal, OMG Document (ad/01-07-08), April 2002
4. UML Testing Profile, Draft Adopted Specification at OMG (ptc/03-07-01), 2003, http://www.omg.org/cgi-bin/doc?ptc/2003-07-01.
5. Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen, and Holger Pals: From Design to Test with UML - Applied to a Roaming Algorithm for Bluetooth Devices (2004)
6. OBSAI_RP1_v1.0; http://obsai.live.visionwt.com/
7. http://www.cheetahtemplate.org/; http://sourceforge.net/projects/cheetahtemplate/
8. Laurent Bartholdi and Zoran Sunik:Some solvable automaton groups (2004)
9. Peter J. Fontana: Effective MDA; Architecture for High Performance Systems; Pathfinder Solutions, (2004)
10. Thomas Bally, Byron Cooky, Satyaki Das, and Sriram K. Rajamaniy. Refining approximations in software predicate abstraction. Proc. of TACAS'04 [Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems], Barcelona (2004)
11. Timed Automata: Semantics, Algorithms and Tools, Johan Bengtsson and Wang Yi. In Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.
12. SOAP Version 1.2 Usage Scenarios, http://www.w3.org/TR/
13. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS 3185.

Design Profiles: Specifying and Using Structural Patterns in UML

Imed Hammouda, Mika Pussinen, Anna Ruokonen, Kai Koskimies, and Tarja Systä

Tampere University of Technology, Institute of Software Systems
{imed.hammouda, mika.pussinen, anna.ruokonen, kai.koskimies, tarja.systa}@tut.fi

Abstract. The concept of a design profile is proposed, integrating UML profiles with pattern-like specification style and generative capabilities. With proper tool support, design profiles allow the checking of design models against architectural rules, and the generation of models based on existing models. Design profiles provide an intuitive way of specifying complex structural relationships and configurations in models. The specification of design profiles using UML class diagrams and existing tool support are discussed. The practical applicability of the design profile concept is demonstrated by specifying the standard design rules of the EJB (Enterprise Java Beans) platform, and by developing a design environment for EJB based on these rules.

1 Introduction

A major trend in software engineering is a shift from code-centric to architecture-centric software development. Architecture is seen as a key asset for explaining the system structure [Joh92], for guiding the system development [JBR99], for reusing software [Bos00], and for assessing the quality of a system in its early development stages [CKK02]. In particular, product-line architectures [CN02] have emerged as an effective means for reducing operational costs through reuse when developing product variants. Companies adopting the product-line approach use and maintain a large set of software assets that are shared by a variety of different products, concurrently developed and reused by different projects. This calls for a well-orchestrated development process and a robust software architecture shared by the product variants.

Even though the importance of software architectures has been acknowledged, proper mechanisms and notations for defining architectural rules and conventions in a systematic manner and tool support linking them to detailed design models have been missing. In principle, the architectural descriptions can be enforced on design models in two ways: design models (or parts of them) can be generated on the basis of architectural models, or existing design models can be validated against architectural models afterwards.

The Unified Modeling Language (UML) [OMG05] has established itself in software industry as a de facto standard for describing software models. Although UML was not specifically aimed at architectural modeling, in practice it has become the most popular architecture description language as well, in spite of its well-known deficiencies in this respect (e.g. [MT00]). Using UML also for architecture design not only allows a

smooth shift from the architecture design to detailed design, but also allows using the same tool environment throughout the design phases.

Profiles are UML's built-in extension mechanism to customize UML to a certain context, purpose, or domain (e.g. the domain supported by a product-line platform). A profile in practice defines a set of stereotypes defining the concepts to be used in the selected context or domain and constraints restricting the usage of these concepts. UML profiles thus provide a mechanism to define architectural rules and conventions relevant in the selected context. In [SeX03], Selonen and Xu present an approach in which they have adopted UML profiles for this purpose. Their profiles are defined in UML class diagram notation, covering both stereotype definitions and structural constraints. The idea is to give the structural constraints in the form of example-like diagrams, which show the allowed relationships with instances of stereotypes. Selonen and Xu further propose tool support for validating design-level models against these profiles.

In this paper, we propose a new type of profile, *a design profile*, which integrates UML profiles with generative properties. This concept is developed with two main usage scenarios in mind: first, it can be used for checking an existing design model against architectural rules specified in the profile, and second, it can be used for guiding the construction of a new design model according to the architectural rules given in the profile. In the former usage scenario, we generalize the approach of [SeX03] by supporting the specification of arbitrary structural model patterns rather than just binary relationships. If a profile is viewed as a specialized modeling language (which it indeed is), the former usage scenario can be seen as syntax checking, while the latter usage scenario can be regarded as an application of the idea of syntax-directed editors. We argue that design profiles could be a first step towards powerful tool support for using domain or platform specific architectural modeling languages based on UML, allowing smooth and consistent transition from architecture design to detailed design.

Design profiles are defined using *design forms*. A design form, given as a UML class diagram, is a description of certain structural properties that must be satisfied by any model given as an instance of a profile. A design profile consists of a set of design forms, and additional specifications concerning a set of design forms rather than a single form. Design forms rely heavily on the notion of aspectual patterns [Ham04, Ham05]. In this paper, we assume that software models are given as class diagrams. In principle, a similar concept could be introduced for other diagram types as well, but this will not be discussed here.

We proceed as follows. In the following section, we briefly review the use of UML profiles for architectural conformance checking, along the lines in [SeX03]. In Section 3, we discuss the generative background of design forms. In Section 4, we introduce the design form concept in detail, and in Section 5, we discuss the current tool support. In Section 6, we show how design forms can be applied for specifying a design language for EJB based applications. Related work is discussed in Section 7 and we conclude with final remarks in Section 8.

2 Architectural Conformance Checking Using UML Profiles

Especially in platform-based software development, the designers are assumed to follow strict architectural rules, constraints, and conventions implied by the platform. Detailed design models that are in conflict with the platform architecture may result in poor performance, hampered maintenance, inconsistent functionality, or simply unrealizable models. Thus, existing design models should be validated against the architectural descriptions. Such conformance checks might result in a list of architectural violations found from the models. These violations should then be considered as indicators for changes required on the models. The changes needed can be made manually or repairing tasks could be generated based on the violation information, enabling a generative and tool-assisted way to repair the models.

Support for architectural conformance checking of existing design models allows the designer more freedom in constructing the designs. It further allows checking the conformance of an existing implementation against the architectural rules, assuming that the code is first reverse engineered to design-level models. In addition, tool support for this approach assumes that the architectural rules and conventions are systematically defined.

In [SeX03], Selonen and Xu present an approach in which they have adopted UML profiles for defining architectural rules and conventions relevant in the selected domain. Their profiles are defined in UML class diagram notation, covering both stereotype definitions and structural constraints. The idea is to give the structural constraints in the form of example-like class diagrams, which show the allowed relationships with instances of stereotypes. Selonen and Xu further propose tool support for validating design-level models against these profiles.

As an example, let us consider a layered architecture designed for a course repository system. The system consists of three layers: a presentation layer that consists of GUI components, an application layer that contains components for the course repository management, and a domain layer containing data storage (i.e. course repository) information. The stereotypes <<Presentation_layer>>, <<Application_layer>>, and <<Domain_layer>>, corresponding to the three layers, are introduced as subtypes of a subsystem, as illustrated in the left most column in Figure 1. It is further assumed that the stereotype <<invocation>> is introduced accordingly as a subtype of a UML dependency. Simplified invocation rules defined for such a three-layered architecture are depicted as an architectural profile in the middle column in Figure 1: presentation layer components can only call application layer components, which, in turn, are allowed to call domain layer components. Namely, the profile is interpreted so that all other dependencies, except of the ones explicitly defined in the profile, are considered to be illegal. While UML profiles can only contain stereotypes, constraints, data types, and tag definitions [OMG05], the associations in Figure 1 (the middle column) are in principle illegal. However, they could be expressed e.g. using OCL constraints, and thus the architectural profile can be normalized into a standard UML profile, as presented in [SeX03, Sel05]. A design level model that is valid with respect of the layering profile is depicted on the right in Figure 1.

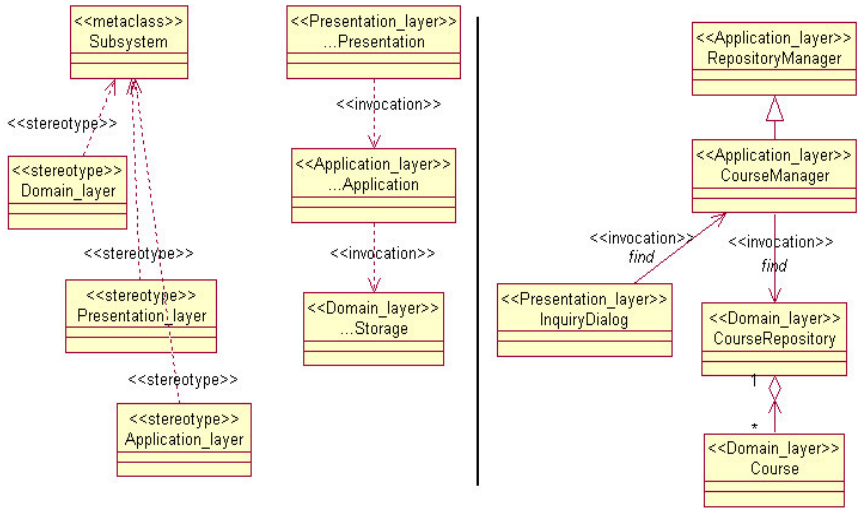


Figure 1. A layering profile on the left and a design level model that is conformant with the profile on the right

UML profiles that use UML class diagram notation are intuitive and easy to construct with any CASE-tool having sufficient support for UML. They further provide the architect and/or the designer a lot freedom in what kinds of rules and constraints she wants to define; it is a matter of the interpretation and application of the profiles how and where the rules are to be applied. For instance, architectural profiles can be used to define global architecture-level rules that are to be applied for the whole design model or more local design-level rules that are to be applied for any selected design fragments.

3 Generative Model Development

In [Ham05], so-called *aspectual patterns* have been introduced as a vehicle for specifying fragments of models for various purposes. Aspectual patterns rely on a general idea of a pattern, which defines a set of roles that can be bound to model elements and a set of constraints which define the required structural relationships and properties of the model elements bound to the roles. Existing tool support [Ham04] allows the specification of patterns and guides a designer to apply the structural composition defined by the pattern in a model.

The pattern concept becomes a generative tool with a simple extension: a role can be associated with the specification of a so-called *default element* that is automatically generated as the element to be bound to the role, if desired. Since the default element can be given as a parameterized template depending on the elements bound to other

roles, a pattern can define a complex generation of new model elements based on existing elements.

As a simple example of a generative pattern, assume that we would like to refactor a design by applying the Facade design pattern [Gam95]. Given a set of classes B_1, ... , B_k and their operations, we want to produce a new model where the services of the B_i classes are accessed through a common class F (for Facade) whose operations are the union of the (significant) operations of the B_i classes. A generative pattern yielding the desired model transformation is given in Figure 2.

In Figure 2, operation roles are marked with dark grey, class roles with medium gray, and association roles with light gray. The dependencies between roles are denoted with arrows, the grey arrows stand for containment relationship. The default elements of roles are here given by specifying the properties that are needed to generate the elements. Note that the properties of other roles can be used in these specifications, as long as the dependencies are followed. Assuming that the left-hand part of the pattern has been bound to actual UML elements (shown on the lower left-hand part of the figure), a model fragment (shown on the lower right-hand part of the figure) can be generated automatically based on the default element descriptions, and the elements of this fragment can be bound to the appropriate roles on the right-hand side of the pattern. In Figure 2, class bindings and some of the operation bindings are shown with broken arrows; association bindings are left out completely to simplify the figure.

A benefit of existing tool support [Ham04] for this kind of pattern concept is that the generation can be carried out stepwise, allowing the designer to intervene in the generation process. In this way a designer may e.g. provide certain elements by hand, or edit a generated element. The tool checks that in all cases the required properties of the bound elements still hold.

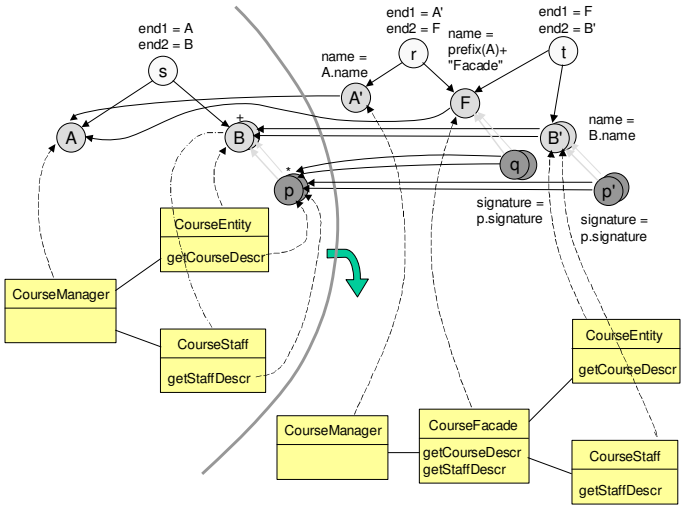


Figure.2. A generative pattern for model transformation

On a high abstraction level, the concept of an aspectual pattern comes close to a UML profile specification, especially when interpreted as in [SeX03]. A profile defines a set of stereotypes, while a pattern defines a set of roles. Concrete model elements are instances of the stereotypes, which corresponds to the bindings of roles to model elements. In both cases, structural constraints associated with roles and stereotypes further limit the legal instantiation of the pattern and profile, respectively.

The unification of patterns and profiles has been studied at conceptual level in [Sel04]. In this paper, we will go a step further and discuss the concrete realization of such unification. An important tool-level benefit of the marriage of profiles and generative patterns is the ability to produce new models under the control of the profile. The checking of models against profiles becomes essentially a pattern detection problem, studied extensively in the context of recognizing design patterns (e.g. [Heu03]). We are currently exploring the application of existing pattern detection techniques for performing automatic binding of roles, given a model and a set of pattern specifications [Wen05].

4 Tool Concepts

In this section, we propose a set of formalisms that we use as tool concepts for achieving the two model development scenarios discussed earlier: architectural conformance checking and generative model development. The conceptual models for the formalisms are given in terms of UML class diagrams.

4.1 Design Profiles

A design profile is a structural entity used to define the architectural rules and constraints for an arbitrary model fragment. Here, we assume that model fragments are expressed as UML class diagrams. Figure 3 depicts the structure of design profiles. A design profile, which is specified as a stereotyped UML package, may contain zero or more sub-profiles. In addition, each profile consists of zero or more design forms, zero or one global forms, and zero or one composition forms. The lifetime of the sub-profiles and forms depends on the lifetime of the parent profile. This property is depicted by the containment relationship shown in Figure 3.

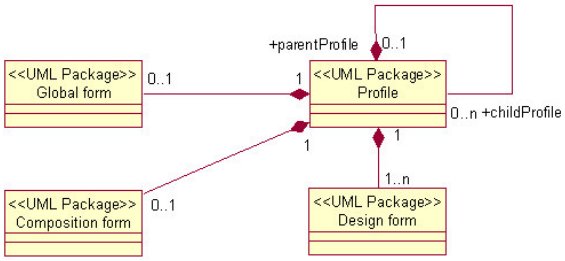


Figure 3. Conceptual model for design profiles

4.2 Design Forms

Design forms represent the core elements of design profiles. Briefly, a design form describes an organized collection of model elements, defining both their properties and inter-relationships.

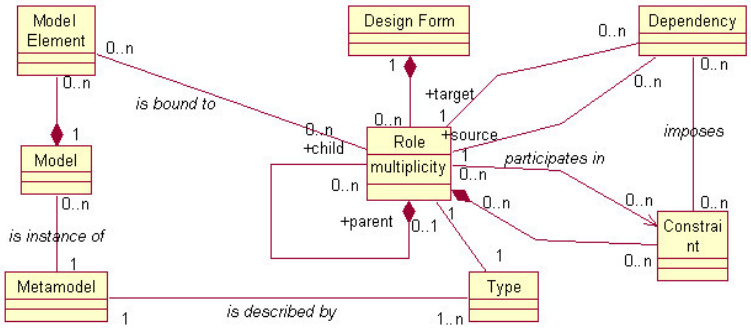


Figure 4. Conceptual model for design forms

Figure 4 depicts a conceptual model for design forms. A design form is a collection of hierarchically organized roles rather than concrete elements. A design form is instantiated in a particular context by binding the roles to certain model elements. Each role can be associated with a set of constraints expressing conditions that must be satisfied by the element(s) bound to a role.

Among other properties, a metamodel is generally assumed to define a containment relationship between the model elements. For example, a UML class may contain a UML operation. In any binding of roles to concrete elements, the containment relationships of the bound elements must respect the hierarchy of the roles.

Furthermore, the metamodels define properties for the model elements that can be checked by constraints. Constraints may refer to the elements bound to other roles, implying dependencies between the roles. For example, a constraint of an association role may require that the association bound to this role must appear between the classes bound to certain class roles, thus implying a dependency from the association role to the two class roles.

A role is associated with a type, which determines the kind of model elements that can be bound to the role. A role type typically corresponds to a metaclass in the metamodel of a given notation. For example, there is a role of type UML class that corresponds to the UML class metaclass in the metamodel for UML. As an example constraint, an inheritance constraint checks the generalization/specialization relationship between two UML classes.

A multiplicity (cardinality) is defined for each role. The cardinality of a role gives the lower and upper limits for the number of elements playing the role in an instantiation of the design form. For example, if a class role has cardinality [0..1], the class is

optional in the design form, because the lower limit is 0. The other possible cardinality values are [1..1] for exactly one concrete element, [0..n] for any number of concrete elements including zero, and [1..n] for at least one concrete element.

4.3 Global Forms

Global forms are used to specify global constraints defined for a group of elements within a design profile. Figure 5 depicts our solution for expressing global forms. For class diagrams, we can define eight possible kinds of elements, whose names are given in form of <<any*>>. For instance, *anyClass* is a reserved term used for specifying global forms. Any design form element of type class becomes associated with the global constraints defined for the element named *anyClass* in the global form. The constraints themselves are specified as a UML note attached to the stereotype kind.

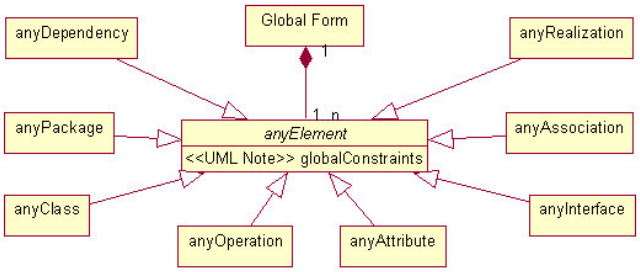


Figure 5. Conceptual model for global forms

4.4 Composition Forms

Composition forms are used to define overlapping relationships between individual roles defined in different design forms. The roles can be at any level in the hierarchy of the design form to which the composition form applies. An overlapping relationship between two roles means that the two roles should be composed dynamically and bound to the same model element. Yet, the two roles still keep their own role views on the model element, e.g. different list of constraints. Figure 6 depicts how overlapping relationships should be specified, i.e. using UML associations to relate the element pairs.

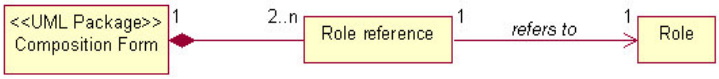


Figure 6. Conceptual model for composition forms

In Section 6, we present concrete examples of the tool concepts discussed above. For exploiting the tool concepts, the next section presents a concrete implementation of a development environment for design profiles.

5 Tool Support - MADE

First, we give a brief introduction to the tool platform. Second, we discuss the main components of the platform architecture.

5.1 Tool Introduction

In order to demonstrate the design profile concept, we use a prototype tool environment known as MADE (Modeling and Architecting Development Environment) [Ham04]. The MADE platform itself is the result of integrating three different tools: JavaFrames [Hak01], xUMLi [Aea02, PS04], and Rational Rose [Ros05]. JavaFrames is a pattern-oriented development environment built on top of Eclipse [Ecl05]. Rational Rose is used as a UML editor. The third component, xUMLi, is a CASE-tool-independent research platform for processing UML models and is used for integrating JavaFrames and Rational Rose. The MADE tool has originally been developed as a stepwise modeling and architecting environment. The tool has been used to manage different kinds of development scenarios [Ham05].

The primary use of the MADE tool has been to synthesize models based on architectural guidelines and rules expressed using a pattern-based tool concept [Hak01]. MADE patterns are role-based structures that can be seen as an isomorphic representation of design forms. In this work, we extend the MADE environment in two ways. First, we exploit the MADE pattern mechanism to achieve a tool infrastructure for representing and applying design profiles. In addition, the pattern composition mechanism, which comes with MADE, can be used to model composition forms. Global forms, in turn, can be expressed as global constraints attached to MADE pattern roles. Second, in addition to the existing generative mechanism, we augment the MADE tool with support for conformance checking functionality.

5.2 Tool Architecture

Figure 7 depicts the architecture of the MADE environment. We distinguish between the following components. For each component, we give the current usage, development, and integration state.

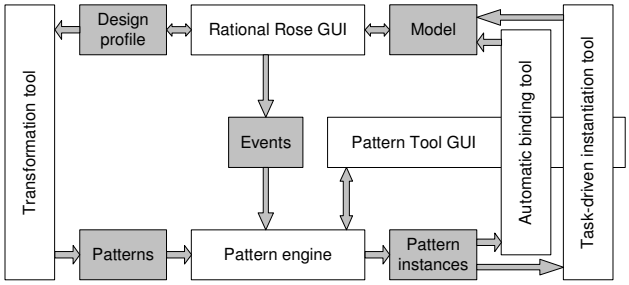


Figure 7. Tool Architecture

Rational Rose GUI. Currently, Rational Rose is being used as the UML editor. The purpose of the CASE tool is two-fold. First, it is used to specify design profiles: design forms, global forms and composition forms. Second, it is used to manage the UML models to which the design profiles are applied. As discussed earlier, this component is integrated into the platform using the xUMLi research tool.

Transformation tool. This component is used to transform the specification of design profiles, modeled as class diagrams, into MADE pattern representation. At this stage, this tool provides support for fully automatic transformation of design forms. Composition forms and global forms, however, are currently transformed manually. This missing feature is being developed.

Pattern engine. This is the core of the tool used to manage the binding process based on the MADE pattern specification. The pattern engine transforms a partially bound MADE pattern into a to-do binding list, i.e. pattern instance. The pattern instance is then given to other tool components, applying the corresponding pattern to the design model under study. The pattern engine updates the to-do binding list every time a binding is established between a pattern role and a model element, In addition, the pattern engine checks that the constraints of the bound roles are satisfied, and generates corrective tasks if this is not the case. For this, this component receives events from Rational Rose.

Pattern tool GUI. This component is used for user interaction with the MADE environment. User interactions can be classified into three main categories: selecting the design profiles to apply, using the selected profiles to generate design models (generative mode), and using selected the profiles to check for conformance of existing models (conformance checking mode). A prototype for this component has been developed and integrated in the MADE environment.

Task-driven instantiation tool. When the generative mode is selected, the pattern engine generates a task for each unbound role that can be bound in the current situation, taking the dependencies and cardinalities of roles into account. Using this instantiation tool, tasks can be performed in two modes. A role can be bound to an existing element, or a default element is first generated according to role specification and then the element is bound to the role. The task list gets updated after a task has been performed, usually creating new tasks. This component is fully functional and integrated in the MADE environment.

Automatic binding tool. When the conformance checking mode is selected, the pattern is automatically bound to an existing design model. In some situations, however, the user may be prompted for certain decisions, for example if the tool cannot decide which model element to consider in a certain binding, among a list of candidate model elements. This component is currently being developed. At this stage, a working prototype has been implemented but has not been yet integrated into the MADE environment.

6 Case Study

We have defined a simple design profile in order to guide modeling of EJB applications. Our *EJB Design Profile* consists of three design forms *EJBLocalEntityBean*, *EJBSessionBean,* and *SessionFacade*, a global form *EJBCommon,* and a composition form *EJBSessionFacade*. The profile hierarchy is presented in Figure 8. Design forms, specifying a certain EJB component, can be applied individually or as a part of a design pattern defined by a composition form. In this case study, we use *EJB Design Profile* in order to model a simple Web service. We will apply a Session Façade design pattern [Alu01] in order to construct a simple Course Repository Web service.

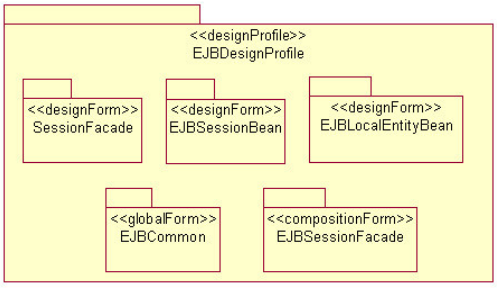


Figure 8. EJB design profile

6.1 Design Forms

For this case study, we have defined design forms to specify container managed Local Entity bean and stateless Session bean components. As an example, Figure 9 shows the Local Entity bean design form. We have used notes (attached to classes) in order to visualize the naming properties and cardinalities and to be able to define them independently of any particular CASE-tool. In practice, to make it more convenient to specify the properties, menu extensions in the CASE-tool could have been used. This, however, might make the properties hidden from the user and might restrict the implementation to be tool-specific.

If the cardinality differs from the default value (1..1), we use the notation *element.cardinality* to specify the cardinality. To allow multiple business methods, for example, we specify *EJBBusinessMethod.cardinality=0..**. We also provide an alternative way to define the cardinalities using extended property page in Rational Rose with the drop-down menu. The menu could be extended also for specifying the naming properties.

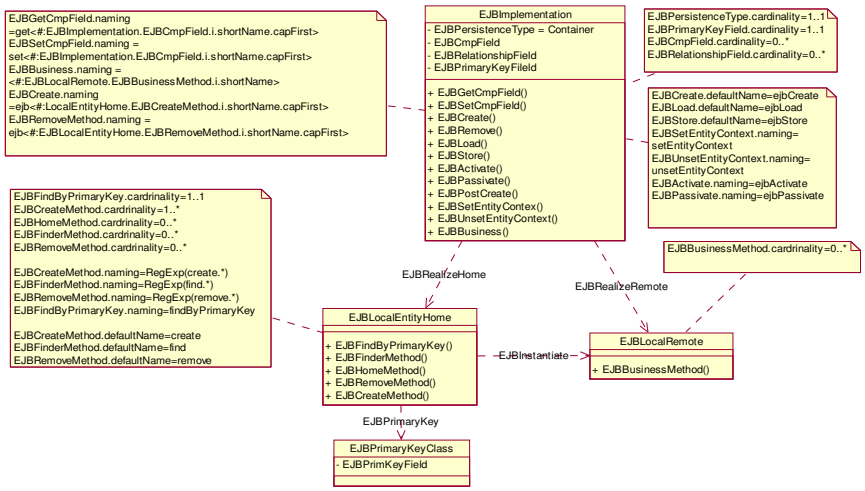


Figure 9. Local Entity Bean

Naming properties can be of fixed values, expressed as regular expressions, or simply refer to elements bound to other roles. In the latter case, the naming property implies that there is an implicit dependency between the roles. The *naming* property defines the allowed values for the element bound to the corresponding role. The value of *defaultName* property is used as a default name for the concrete elements when they are generated.

The EJB 2.1 specification [EJB2.1] defines several naming restrictions for EJB methods. In order to illustrate this in design forms, consider home method *create*, defined in the local home interface. This method must start with the prefix 'create'. In addition, the method must have a corresponding bean method in the implementation class and should be prefixed with 'ejb'. In the design form specification, the methods are represented using *EJBCreateMethod* in *EJBLocalEntity* Home interface and *EJBCreate* in *EJBImplementationClass* implementation class. The corresponding naming rules can be expressed using naming constraints EJBCreateMethod.naming = RegExp(create.*), attached to *EJBLocalEntityHome* interface, and EJBCreate.naming = ejb<LocalEntityHome.EJBCreateMethod.i.shortName.capFirst> attached to *EJBImplementationClass* class. The latter constraint refers to role instance of *EJBCreateMethod* and implies that for each method instance a corresponding method in the implementation class must be generated. For the generated method, naming constraint is used as default value overriding a possible *defaultName* defined. A role reference is always denoted by the notation <*referredRoleName*>.We will also use a function to capitalize the first letter of the *EJBCreateMethod*, as required in the EJB specification. In addition to these constraints, a default method name 'create' has been attached. In the same manner, we can create e.g. getter and setter methods for any attribute role with any cardinality. As an example, we have *EJBCmpField* with cardinality 0..* in *EJBImplementationClass* (Figure 9).

In some cases it might be useful to define more restricting constraints e.g., parameter constraints or requiring the operation signatures to be equal. This might be essential for more efficient use of the EJB design profile. In the scope of this example, however, we want to illustrate the main concepts of the design profiles as well as show how they can be used in order to capture design rules of Enterprise Java Beans.

6.2 Global Forms

To define design profile specific constraints we have defined one global form, *EJBCommon*. We have attached a naming constraint to *anyPackage* role, presented in Figure 10. The constraint is for ensuring that all the instantiated packages should start with prefix 'Course', 'Staff' or 'Student'. By adding the global constraint, we want to restrict the instantiated applications and their subsystems to use these domain-specific concepts.

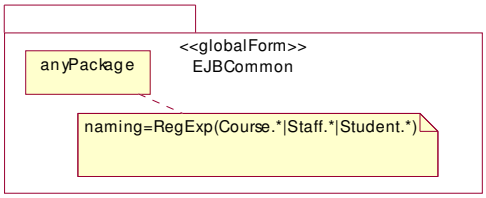


Figure 10. Global form

6.3 Composition Forms

Session Façade, as represented in this example, is concerned with the component relationships and the pattern roles are bound to a certain component, a subsystem, not separating the remote and home interfaces. In Figure 11, the package on the left hand side shows the form representation of Session Façade. The cardinalities define that multiple elements can be bound to *businessComponent* and *businessEntity* roles. Session Façade design pattern is platform-specific implying that its implementation has a connection to a certain components in the EJB platform. s*essionFaçade* role is supposed to be bound to a stateless session bean and a component bound to *businessEntity* role should be a local entity bean. *businessComponent* role can be bound to any business components from the profile or any arbitrary class. If an arbitrary class (not defined in the design profile) is bound to the role no additional rules are related to it.

The package on right hand side in Figure 11 defines the composition form for Session Facade. The composition rules are defined as role references. All the role references are expressed as classes, where the prefix of the name defines the design form and the postfix defines the role name. The dependency between two role references defines overlapping roles. Composition form *EJBSesionFacade* defines two composition rules: (1) *sessionFacade* role of *SessionBean* design form is overlapping with *EJBSessionBean* role of the *EJBSession* design form (2) *businessEntity* role is overlapping with *EJBLocalEntityBean* role of the *EJBLocalEntity* design form.

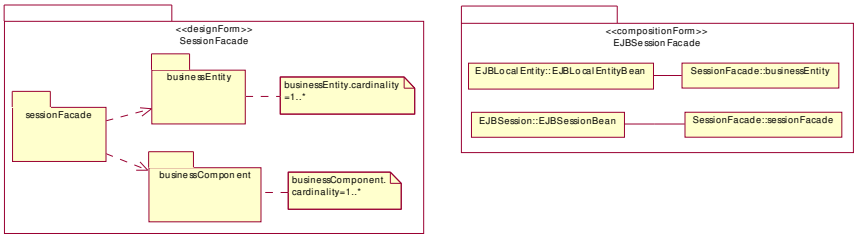


Figure 11. Session Façade design form and composition form

The Session Façade design pattern is constructed as a composition of the three above mentioned design forms: *EJBLocalEntityBean*, *EJBSessionBean* and *SessionFacade*. Composition rules for patterns are defined textually using pair of pattern roles (Figure 12). In addition to overlapping roles, pattern composition rules also define that *SessionFacade* must be applied first. When the *SessionFacade* is applied and the overlapping roles are bound, new tasks will be generated. For example, when *sessionFacade* package role is bound, the user gets a new task to bind the same package to *EJBSessionBean* role. Therefore, only one package is created and bound to both roles.

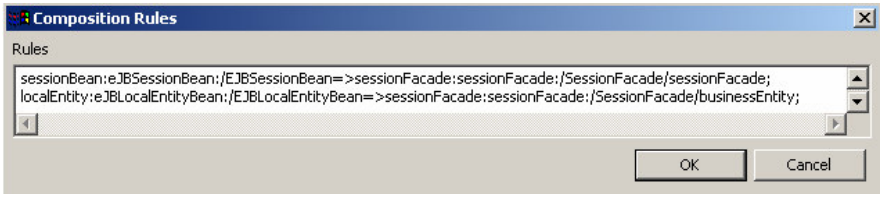


Figure 12. Pattern composition in MADE

6.4 Applying EJB Design Forms

In the current user interface *Apply form* and *Show applied forms* actions are available (of Figure 13 left hand side). Each applied design form should have a target package and a target diagram. The target diagram can be specified by 'Select location' - dialog (Figure 13). The target package for the generated form instance is the parent package of the selected diagram. After selecting the target, a new design form instance is created and opened up into the INARI task window (on the right Figure 13). By selecting the option '*Show applied forms*', the user gets a dialog showing all design form instanced applied in the current package.

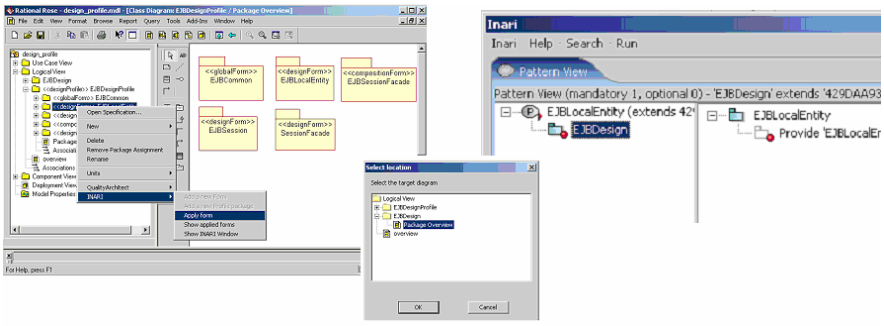


Figure 13. Applying design forms

We have applied the *SessionFacade* composition pattern for constructing a simple Course Repository Service. The Session Façade *CourseService*, implemented as a stateless session bean, provides the service endpoint and encapsulates the business components. The business components, *CourseEntity* and *StaffEntity*, container managed entity beans, handles and stores the course-related data. We have added an additional value object class *Course* to enable efficient data transferring [Alu01]. On the left hand side of Figure 14, the *CourseService* package represents the package structure of the application. The *CourseFacade* package, on the right hand side, shows the internal structure of the *CourseFaçade* component.

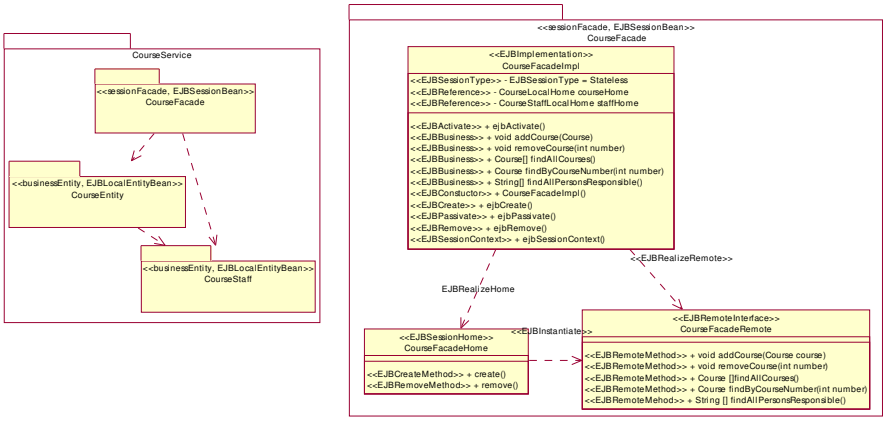


Figure 14. Course Repository Service

The elements instantiated from the design profile will automatically get the same stereotype name as the corresponding role name. A stereotype indicates that a model element is bound to a certain role and that there are some rules applied to it from the design profile. In this example, profile stereotypes start with a capital letter and pattern roles with a lower case e.g. <<*EJBEntityBean*>>, <<*businessComponent*>>. When an entity bean element is bound to a business entity role, multiple stereotypes are used and are given, for example, as <<*EJBEntityBean, businessEntity*>>. When an entity

bean is bound to a pattern role, there are two sets of validity rules to be satisfied: internal rules from the component design form and external rules from the Session Façade design pattern that is applied. Both rules, however, can be violated when the model is modified. The recorded binding information can then be used to maintain the model by generating a new repairing task for every violated rule. The task is shown in the INARI task window (on the right in Figure 13).

7 Related Work

An approach for validating UML design models against architectural rules and conventions, given as *architectural profiles*, has earlier been proposed by Selonen and Xu in [SeX03]. Architectural profiles are extended UML profiles specialized for describing architectural constraints and rules for a given domain [Sel05]. The structural constraints and rules are given e.g. in a form of a class diagram, as in design forms. A set of *conformance rules* are then used to check whether a given UML model conforms to those constraints and rules. The design models to be validated are annotated with the stereotypes defined in the profile. This enables the conformance operations to identify which architectural rules are to be applied for each model element. This approach has been implemented in artDECO toolset [Aea02, PS04], which allows the conformance checks to be run and configured. In addition, it allows listing the errors found and browsing their sources in the models. The approach by Selonen and Xu and artDECO tool has been applied when maintaining a large-scale product platform architecture and real-life product-line products built on top of this platform [Riva04].

Conformance checking using design forms follows the ideas presented by Selonen and Xu. We aim at providing support, similar to using the error browser of artDECO toolset, for the designer to manually browse and act on the violations. In addition to that, application of design forms allows the generation of repairing tasks based on the violation information. The designer can then repair them by using the design form in a generative way.

Generative approaches for constructing running systems have been commonplace since the early days of computing. The most recent manifestation of generative approaches is MDA (Model-Driven Architecture, [OMG03]), where platform-dependent system models are generated from higher-level models using various model transformation techniques. In principle, MDA and its accompanying technologies can be regarded as a framework for model-based software development, analogous to compiler technologies of conventional programming languages. However, compiler technologies are a mature, well-established area both theoretically and in practice, whereas model transformations are in their infancy. Design profiles could be seen as a first, modest step in developing the counterpart of context-free grammars for model-based software development.

UML profiles introduce the ability to tailor the UML metamodel for different platforms or domains. Enterprise JavaBeans (EJB) technology is the server-side component architecture for J2EE (see EJB 2.1 [EJB2.1]). A UML Profile for EJB [JSR26] defines standard mapping between EJB architecture and UML. The profile presented in [JSR26] is, however, out of date since it supports EJB specification version 1.1. The UML 2.0 Superstructure [OMG05] specification gives example component profiles for

J2EE/EJB, .NET, COM and CORBA. They only define common stereotypes and are provided as illustration of how UML can be customized. The EJB design profile, described in this paper, is constructed based on these three specifications.

8 Discussion

We have proposed a new kind of a UML profile, a design profile, for specifying structural rules for models expressed as UML class diagrams. Typically, such rules capture architecture-level decisions or their implications. We have shown that a design profile can support two central usage scenarios: checking a model against architectural rules, and tool-supported guidance of creating architecturally correct models. A prototype tool environment supporting design profiles has been implemented; however, the automated pattern detection support required by the first scenario is still under work [Wen05].

The idea of design profiles can be generalized in a straightforward way for other than class diagrams. For example, if an architecture implies certain behavioral rules, design profiles can be given for behavioral models given as sequence diagrams. In that case the design form notation is naturally a sequence diagram as well. However, a particular diagram type may require specific conventions for specifying design forms, characteristic to the diagram type. For example, in the case of sequence diagrams it may be necessary to specify that certain messages cannot happen in a certain context, or that certain messages have to take place in a particular order, with arbitrary intervening messages. Such rules are difficult to express as sequence diagrams without adopting special conventions. We are currently investigating the required notation for expressing behavioral design profiles as sequence diagrams [Kos05].

Acknowledgements

This research has been financially supported by TEKES, Nokia, Plenware Group, Solita, TietoEnator, Plustech, and GeraCap.

References

[Aea02] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä: xUMLi: Towards a Tool-independent UML Processing Platform, In *Proc. the 10th NWPER Workshop*, pages 1-15, IT University of Copenhagen, Denmark, 2002.

[Alu01]D. Alur, J. Crupi, and D. Malks: *Core J2EE Patterns, Best Practises and Design Strategies*. Sun Microsystems Press, A Prentice Hall Title, 2001.

[Bos00] J. Bosch: *Design & Use of Software Architectures: Adopting and evolving a product-line approach*, Addison.Wesley, 2000.

[CKK02] P. Clements, R. Kazman, and M. Klein: *Evaluating Software Architectures – Methods and Case Studies*, Addison-Wesley, 2002.

[CN02] P. Clements and L. Northrop: *Software Product Lines: Practices and Patterns,* Addison.Wesley, 2002.

[Ecl05] Eclipse WWW site, 2005. At URL http://www.eclipse.org.

[EJB2.1] Enterprise JavaBeans TM Specification, Version 2.1. Sun MicroSystems 2005. At http://java.sun.com/products/ejb/docs.html.

[Ham05] I. Hammouda: A Tool Infrastructure for Model-Driven Development Using Aspectual Patterns. In Sami Beydeda, Matthias Book, and Volker Gruhn, eds., *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Hak01] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa: Generating Application Development Environments for Java Frameworks. In *Proc. GCSE 2001,* pages 163–176, Erfurt, Germany, September 2001. Springer, LNCS 2186.

[Ham04] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen: Adaptable Concern-based Framework Specialization in UML. In *Proc. ASE 2004*, pages 78–87, Linz, Austria, September 2004.

[Heu03] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe: Automatic Design Pattern Detection. In *Proc. IWPC 2003*, pages 94-103, Portland, Oregon, USA, 2003

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh: *The Unified Software Development Process*, Addison-Wesley, 1999.

[Joh92] R. Johnson: Documenting Frameworks Using Patterns. In *Proc. OOPSLA '92,* pages 63-76, Vancouver, Canada, October 1992.

[JSR26] UML/EJB Mapping Specification: UML Profile for EJB. Rational Software Corporation, 2005. At http://jcp.org/aboutJava/communityprocess/review/jsr026/.

[Kos05] J. Koskinen, K. Koskimies, T. Mikkonen, and T. Systä: Behavioral UML Profiles and their Application, 2005. Manuscript, submitted.

[MT00] N. Medvidovic, R.N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE* 26,1, pages 70-93, January 2000.

[OMG03] MDA guide version 1.0.1, 2003. At http://www.omg.org/.

[OMG05] The Object Management Group: *Unified Modeling Language Specification*, version 2.0, May, 2005. At http://www.omg.org/uml/.

[PS04] J. Peltonen and P. Selonen: An Approach and a Platfrom for Building UML Processing Tools, In *Proc. WoDiSee'04 workshop of ICSE'04*, pages 51-57, 2004.

[Riva04] C. Riva, P. Selonen, T. Systä, and J. Xu: UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance, In *Proc. ICSM'04*, 2004.

[Ros05] Rational Rose WWW site, 2005. At http://www.rational.com/products/rose/index.jsp.

[Sel03] P. Selonen: Set Operations for the Unified Modeling Language, In P. Kilpeläinen and N. Päivinen (eds.), In *Proc. SPLST'03*, pages 70-81, Kuopio, Finland, June, 2003.

[SeX03] P. Selonen and J. Xu: Validating UML Models Against Architectural Profiles. In *Proc. ESEC 2003*, pages 58-67, 2003.

[Sel05] P. Selonen: *Model Processing Operations for the Unified Modeling Language*, doctoral dissertation, Tampere Univ. of Tech., Finland, 2005.

[Wen05] S. Wenzel: Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams, 2005. Manuscript, submitted.

Visualizing and Comparing Web Service Descriptions in UML

Juanjuan Jiang, Juha Lipponen[1], Petri Selonen, and Tarja Systä

Tampere University of Technology, Institute of Software Systems
juanjuan.jiang@tut.fi, juha.ta.lipponen@nokia.com,
{petri.selonen, tarja.systa}@tut.fi

Abstract. Web services are described and located using the Web Service Description Language (WSDL). While considerable tool support is available for generating WSDL descriptions from existing service interfaces, these tools may differ in how the generation is done. During the evolution of the service, its interface may also change, requiring the generation of a new WSDL description as well. Consequently, it is important to understand the WSDL descriptions and to be able to compare them. In this paper we present an approach to analyze and compare WSDL descriptions at UML level. We show the applicability and usefulness of the approach with two case studies. We first use the approach to compare tool support available for constructing Web services in terms of WSDL descriptions they generate for the same example Web service. In the second case study we explore and compare two different versions of a particular Web service and its interface.

1 Introduction

Technologies supporting the construction of distributed systems, such as CORBA, DCOM, and RMI have traditionally been used for system-to-system communication. However, these technologies have not completely reached the goal of allowing any systems to communicate with each other. Instead, these technologies are more or less middleware, platform, or language dependent. The next step towards this goal was the introduction of the Web service concept. Web services are software systems designed to support interoperable machine-to-machine interaction over a network [23]. They aim at making the information exchange easy by providing a way to integrate systems loosely and independently from the platforms and programming languages used. In the last couple of years the interest towards Web services has rapidly increased, both from the research and from the business perspective.

Web services use XML-based standards such as Web Service Description Language (WSDL) [24] and Simple Object Access Protocol (SOAP) [25]. WSDL is an XML format for describing network services as collections of communication endpoints capable of exchanging messages. It is the current standard way to describe and locate Web services. Thus, WSDL can be seen as a key for Web service interopera-

[1] Currently working at Nokia

bility. The cross-platform interactions, in turn, are handled using SOAP. Although SOAP and WSDL are still under development, they have been generally accepted as the basic technology to be used in Web services. Even though there is a general agreement on these standards, many challenges and problems still exist. First, these standards are constantly evolving, yielding to e.g. version problems. Second, the specifications contain many open and optional issues, which allow a variety of different ways to use them in implementations.

A lot of tool supports are currently available to construct Web services. Many Web service toolkits allow automatic generation of WSDL descriptions from existing interface implementations to be exposed to Web service clients. These approaches are used especially to create RPC-style (Remote Procedure Call) interfaces for Web services. In these approaches the designer does not really have a control on how WSDL descriptions are constructed. Furthermore, when WSDL descriptions are automatically generated, they often simply reflect the existing (e.g. Java) interface and do not really consider the client side. Even though being an easy way to develop the interface definition, the costs come with the inflexibility: the evolvement of service interface often requires changes in WSDL documents, which in turn may require changes in clients' ends. Thus, it is important to understand the WSDL descriptions and to be able to detect changes in them. Moreover, the tool support available varies in how the WSDL generation is done. For instance, the bindings to the transport mechanisms or types used in operation descriptions may vary. Therefore, it is useful to be able to compare the different tools with respect of how they generate the WSDL descriptions.

In this paper[2] we present an approach that provides UML-based [20] support to visualize and compare WSDL descriptions. This is enabled by tool support for transforming WSDL documents to UML representations [8] and by applying set operations for comparing UML diagrams pair-wise [21,17]. Using the UML class diagram notation, the service descriptions are easy to comprehend by a software engineer. The results of the application of the set operations are also visualized as UML class diagrams: colors are used to highlight the shared parts and parts that belong to one model but not the other.

The proposed approach can be used e.g. to compare Web service tool support. We present a case study, in which we have developed the same service using four different commonly used toolkits. We will show that even in the case of a relatively simple Web service, the generated WSDL documents vary. A short summary of the approach is characterized by the authors in [7].

The proposed approach can also be used to analyze the evolution of a Web service or changes due to supporting different versions of Web service standards (WSDL and SOAP) or different binding mechanisms used. We have used the approach to compare two WSDL documents generated from two versions of the same Web service. Since changes in WSDL documents may require changes in the client side, it is important for the clients to know what has been changed in the WSDL document. Our approach provides a way to visualize these differences. The engineer can then easily recognize the points of possible changes and conclude whether changes in the client side are needed.

[2] This paper is an extension of [7]

2 Approach and Tool Support

The introduced tools are built on top of xUMLi [1,15], a CASE-tool independent software platform supporting the development of various kinds of UML processing facilities. It allows users to build arbitrary UML model processing operations and to combine them using a scripting mechanism to construct more complicated operations. The tools are implemented as xUMLi compliant COM components.

The workflow for comparing different Web services descriptions at UML level starts from transforming WSDL documents to UML views. These views contain redundant UML data in most cases; for every occurrence of a certain WSDL element type (e.g. port) a new class representing this instance is created. Thus, these views are abstracted to efface instance information and to capture the logical structure of the documents. Comparisons between the views and/or abstract views are then carried out using set operations that identify and visualize the differences in them. The resulting differences are finally exported to a UML CASE-tool. We currently use Rational Rose. These differences between UML models are distinguished by different colors in Rose. Exploring the UML diagrams, users can easily conclude differences between WSDL documents.

2.1 UML-based Reverse Engineering of WSDL Documents

A WSDL document in the proposed approach is first transformed into a UML class diagram using an extended UML profile that describes the structure of WSDL documents [8]. The profile defines the types of elements that may occur in WSDL documents using stereotypes. The allowed relationships among the elements are also defined in this profile. All the stereotypes, except of «Extension», represent key elements specified in WSDL, SOAP, XML Schema, and WS-I specifications. Stereotype «Extension» is designed for extensible elements that are currently not defined in the profile. Elements in WSDL document are modeled as classes in the UML class diagram, and attributes of the elements are placed as attributes of the corresponding classes. The stereotypes represent the qualified names of the elements and are concluded from the class name and the namespace definition, following the WSDL standard and WS-I profile practices. For instance, even though any namespace prefix can be used and the choice is not semantically significant, WS-I Basic Profile uses prefix "wsdl" for namespace *http://schemas.xmlsoap.org/wsdl/*. Finally, the aggregation relationships are formed based on element-subelement relationships in the WSDL document.

As an example, let us consider a particular Web service *InstantMessageAlert* by BindingPoint to explain how WSDL document is transformed to UML models. This service is for sending instant messages. The WSDL document of a free version of *InstantMessageAlert*[3] Web service allows up to 3 messages per device per day. The generated view (UML class diagram) contains quite a lot of classes (263) and associa-

[3] Available at http://www.bindingpoint.com/ws/imalert/imalert.asmx?wsdl

tions (262). To save space, Figure 1 presents a corresponding UML class diagram only including *service* and *port* content with some of the attributes omitted.

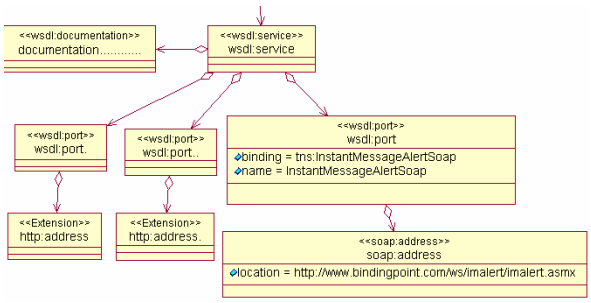


Fig. 1. A partial view of a WSDL of InstantMessageAlert service

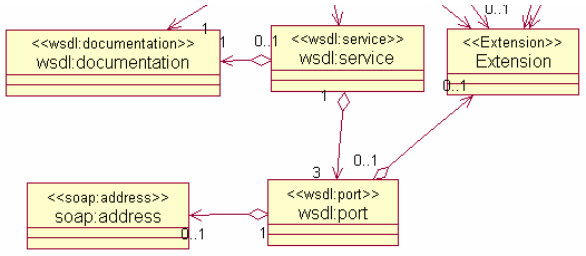


Fig. 2. A part of an abstract view for a WSDL of InstantMessageAlert service

Since this approach uses stereotypes specified in the WSDL profile, the view typically has many classes with the same stereotype. For instance, in Figure 1, there are several classes that all have a stereotype «wsdl:port». The abstraction operation aims at constructing a class diagram that no longer focuses on these individual classes, but rather illustrates what kinds of classes (identified by stereotypes) are used and how many times and how these different types of classes relate. Three principles are used when composing an abstract view of the (possibly reverse engineered) detailed UML class diagram representation of the service description. These principles are applied in consecutive steps:

i. classes having the same stereotypes are mapped to one class with the shared stereotype;
ii. associations are mapped with the same association in the abstracted view, if the corresponding ends of the associations are joint according to the principle in step (i);
iii. multiplicities of associations are defined according to the times of appearances of the associations mapped to it;
iv. the stereotype is changed to equal the class if needed; and
v. attributes are removed.

Following the rules of abstract operations, the view in Figure 1 is transformed to

an abstract view shown in Figure 2. The entire abstract view includes 27 classes and 33 associations. Comparing with the low-level view, the abstract view is compressed and reduced by 236 classes and 229 associations.

2.2 Set Operations on UML Diagrams

UML diagrams can describe a system from different perspectives, on different levels of abstraction, and at different stages of software development process, making them dependent on each other. The dependencies can be exploited using UML set operations [17]. Set operations like union, intersection, and difference are binary operations that on the basis of two UML diagrams of a particular type produce a new UML diagram of that same type. The connectivity properties of the operations allow them to be combined into more complex expressions (e.g. symmetric difference). The operations can be used for merging the modeling artifacts produced by several designers and to support incremental software development. They also provide a mechanism for composing and decomposing of models according to different concerns. Moreover, they provide a way to improve model comprehension as the designers and stakeholders can compare and slice models according to different viewpoints, especially when accompanied with suitable visualization tools (e.g. coloring).

The set operations are based on deriving a correspondence relationship among elements that are seen to represent the same semantic concept. Correspondence is used as a criterion for resolving the possible conflicts between the model elements (e.g. inconsistent multiplicities), and as a basis for performing the actual operation. Obvious correspondence criteria include the type (metaclass) and name of the model elements, and a repository identifier when available. In addition, the context of the model elements specifying their semantics is required to be corresponding. The context can comprise, for example, the end elements of a relationship, or a composite element (e.g. class) of part elements (e.g. attribute).

The basic definitions offer a starting point for applying whatever rules and heuristics might be available for the given domain. While the correspondence definitions can be arbitrary complex, in practice the way humans use identifiers makes names and types of the model elements a very useful correspondence criterion. The described approach works on static models, and static structure models in particular. Depending on the naming scheme used, it can be also used on instance-level structure diagrams.

In the context of this study, the operations are used for comparing the UML models representing different WSDL documents against each other. By visually highlighting their intersection and difference against their union, the user is provided an intuitive way for observing the commonalities and variations between the models. A concrete implementation for the operations is presented by van der Ven [21]. The set operations have been earlier applied when comparing UML-based reverse engineering methods provided by different CASE-tools [9].

When applying the set operations for UML-level Web service descriptions, the correspondence relationship is defined as follows:

i. two classes are corresponding, if their names are matching (string comparison), and

ii. two aggregation relationships are corresponding, if their end elements (aggregating and aggregated classes) are corresponding.

Note that two classes are matched only by their names, not considering their stereotypes. Since the abstraction operation changes the stereotype to equal the class name if needed, the set operations might find more matches when applied to compare abstract level model than when applied to compare low-level models. An example of this is shown later. Note also that in the current version the multiplicities of aggregations are ignored.

3 Comparing Web Services Tool Support

In our first case study we compare popular toolkits supporting the construction of Web services by implementing the same example Web service using four different toolkits and by comparing the generated WSDL documents. Tools used in this case study all support, at least, two basic methods to create Web services. Developer can point out the interface functions of a Web service and then use a tool to generate the WSDL document. This method is referred to as "Code to WSDL" in the sequel. Alternatively, she can write manually the WSDL document and then use a tool to generate the necessary code for a Web service. This method is referred to as "WSDL to code" in the sequel. "Code to WSDL" is usually a better choice for developers who are unfamiliar with Web services, since it is an easy and fast method and the developer does not need to be familiar with details of WSDL. However, "code to WSDL" has also downsides that may cause interoperability problems. When WSDL descriptions are automatically generated, they often simply reflect the existing code interface and do not really consider the client side. Comparing with "code to WSDL", "WSDL to code" is a safer choice since the developer has the control on the construction of the WSDL description. Therefore, such interoperability problems are less likely to be created by accident. However, this approach is typically less efficient from the designer's point of view. Moreover, she should have enough knowledge for constructing WSDL documents. In this case study, the "code to WSDL" approaches supported by chosen Web service toolkits are compared. The WSDL documents they generate for the same example Web service are compared and the differences are analyzed.

3.1 Tools Used in the Case Study

Table 1 shows the tools used in the case study. All the toolkits have been available for some time, and they all follow basically the same way of constructing WSDL documents when using the "Code to WSDL" approach. IBM's and Oracle's tools use components made by Sun Microsystems, and both of them use Java as their programming language. When using Microsoft's toolkit, the developer can choose from

multiple programming languages. For this case study C# was chosen. WSDL documents were generated using the default options of these tools.

Table 1. Tools used in the case study

Tool	Developer	Version	Progr. lang.
.NET Framework [11]	Microsoft	1.1	C#
Java Web Service Developer Pack [19]	Sun Microsystems, Inc.	1.3	Java
Application Server Containers for J2EE 10g [13]	Oracle	1.0.3.0 (developer preview)	Java
WebSphere SDK for Web Services [6]	IBM	5.1	Java

3.2 Implementing a Sample Web Service

JWSDP, WSDK and OC4J offer basically same kind of support for "code to WSDL" method. The developer first constructs the interface class of the Web service. Based on the information on the interface class, a tool then generates a WSDL document and the binding classes for the server program and the implementation of the Web service. .NET has a bit simpler "code to WSDL" process from the point of view of the developer. She only points out the functions (by adding a [WebMethod] mark on them) to be included in the interface of the Web service. The WSDL document itself is generated dynamically by a request of a client.

The implemented Web service gives a client a chance to get information about events that can happen in the stock market. The Web service needs the following information from the client: an event (e.g. rate of a certain stock) and a connection (e.g. email). After this the Web service checks once in a while, if the event has occurred. If it has, the connection is used to inform the client.

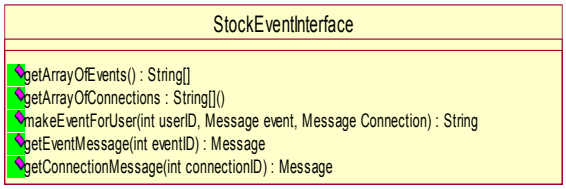


Fig. 3. An interface of the implemented Web service

Figure 3 shows the interface of the Web service. The functions available have varying return and parameter types, namely, string, integer, and arrays. There is also a developer-defined bean type custom class (Message in the interface in Figure 3), which is a class with get and set operations. This way we hoped to find as many of the differences that the toolkits make when constructing the WSDL document as possible. At the code level, the only difference among the implemented Web services is that .NET did not add the set and get operations for the variables of the custom

class to the WSDL document. Instead, the developer has to define the custom class variables public if she wants them to be included in the WSDL document.

3.3 Differences in WSDL Documents

Table 2. Class-level differences in abstract views

Tool	.NET	JWSDP	OC4J	WSDK
.NET	0	1	1	1
JWSDP	4	0	3	4
OC4J	1	0	0	1
WSDK	0	0	0	0

The generated WSDL descriptions are transformed to UML class diagrams (views) using WSDL2UML operation. They are further abstracted (abstract views) to visualize the logical structure of the descriptions in a more compact form. Comparing different WSDL documents follows the workflow described in section 2. The set operations highlight the discovered differences when applied to two views or two abstract views. Crosschecking of the results of the application of the set operations on the abstract views are listed in Table 2. The amount of different classes (i.e., elements in the WSDL document) found are listed so that a number of a cell indicates how many elements are found from the abstract view constructed from a WSDL document generated by a tool naming the row but not found from the abstract view constructed when using a tool naming the column. For example, the cell whose row is named "JWSDP" and column is named ".NET", means that there are four elements in the WSDL document generated by JWSDP tools (called WSDL_JWSDP in what follows), which are not in WSDL document generated when using .NET toolkit (called WSDL_.NET in what follows).

Fig. 4 shows the merged abstract views generated when using JWSDP and .NET toolkits. Green elements (middle grey in the picture) are common to both of the abstract views, red elements (dark grey in the picture) are not included in WSDL_.NET, and yellow elements (light grey in the picture) are the ones that are not included in WSDL_JWSDP. There are obviously also differences in aggregations; the differing classes (red and yellow) are aggregated to other classes. For simplicity, the multiplicities are left out.

WSDL_JWSDP has four elements that do not occur in a WSDL_.NET. Three of these elements, namely xs:complexContent, xs:restriction and xs:attribute, are generated to help the client side developer by providing some attributes, for example arrays, in a way that they can be used directly, not via a custom class. When using the other toolkits, the client side developer has to get the array attribute from a custom class with a get operation, which makes programming a bit more complicated.

Oracle and JWSDP toolkits added an import element, which was not done by the other case study tools. This element was used to import the SOAP encoding (http://schemas.xmlsoap.org/soap/encoding/). .NET toolkit added this encoding in the wsdl:definition element, but the WSDK toolkit did not add the encoding.

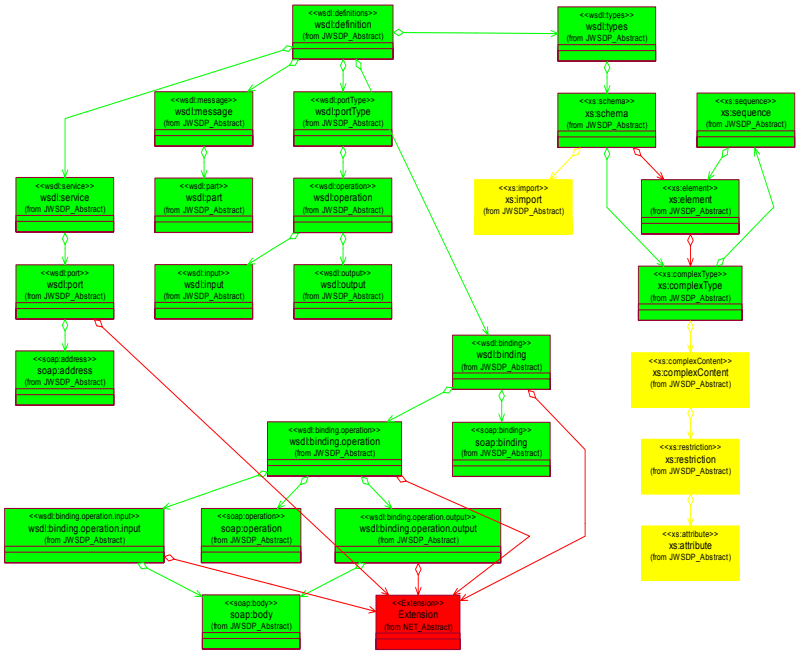


Fig. 4. An abstract UML model illustrating the differences in the WSDL documents generated by JWSDP and .NET toolkits

WSDL document generated by the tool that comes with the .NET toolkit included something extra, which was shown in the abstract model as Extension element. When looking at the concrete model we found out that .NET toolkit adds SOAP 1.2 support automatically. This can be concluded when applying the set operations to low-level views. A part of the views is shown in Figure 5. The coloring is the same as one used in Figure 4. Only two corresponding classes were found: soap:address and wsdl:definitions. The rest of the elements belong only to the WSDL document generated by .Net tools. The reason for this is the class names in these low level models differ, mostly by the prefixes used. Other case study tools did not in charge of SOAP 1.2, at least not automatically. Thus, Web services constructed by .NET can be accessed with both 1.1 and 1.2 versions of SOAP.

4 Analyzing WSDL Evolution

In our second case study we analyze and compare WSDL documents composed for two versions of *InstantMessageAlert* Web service by BindingPoint; a free version (*InstantMessageAlert*) and an advanced version requiring a license key (*InstantMessageAlert Pro*). We applied our approach to study what kind of and how many differ-

ences exist in the WSDL descriptions. The free version has been introduced in Section 2.1 and a part of the view is shown in Figure 1. The advanced version[4] allows up to 3000 messages per device per month.

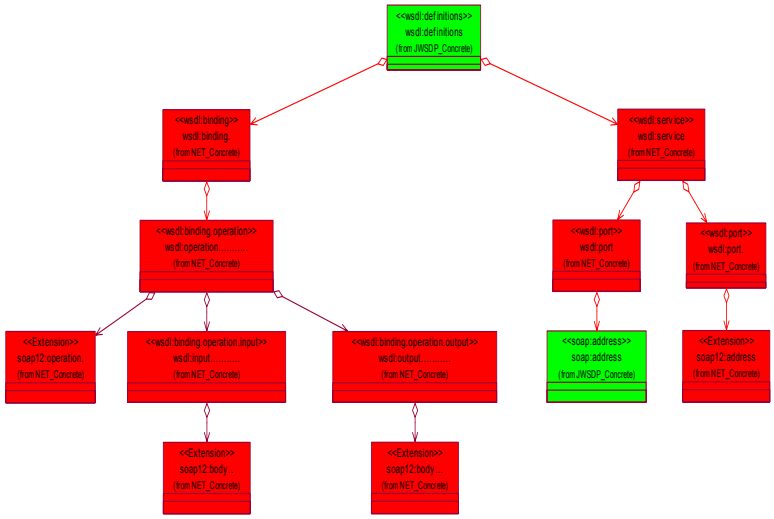


Fig. 5. Part of the concrete UML -model of the WSDL document generated by the tool that comes with the .Net

Table 3. Attribute differences discovered in WSDL documents of InstantMessageAlert and InstantMessageAlert Pro

Classes with attribute differences	**Free version**	**Advanced version**
wsdl:port	name="InstantMessageAlertSoap"	name="InstantMessageAlertPro-Soap"
wsdl:port	binding="tns:InstantMessageAlertSoap"	binding="tns:InstantMessageAlertProSoap
soap:address	location="http://www.bindingpoint.com/ws/imalert/imalert.asmx"	location="http://www.binding-point.com/ws/imalertpro/imalertpro.asmx"

Obeying the rules used in the transformation of the free version, the advanced version is transformed to a view and an abstract view. Comparing abstract views by

[4] located at http://www.bindingpoint.com/ws/imalertpro/imalertpro.asmx?WSDL

applying the set operations, we did not find any differences in classes nor associations. We further compared the views to detect detailed differences. As a result, set operations highlight classes in which attribute differences occur. Table 3 lists the attribute differences existing in classes wsdl:port and soap:address, meaning that there are attribute differences in the corresponding elements in the WSDL documents.

Let us assume a scenario, in which the client side uses currently the free version but intends to employ the advanced version. An interesting question in this case is: what kinds of conditions require client applications to be modified when sending messages to *InstantMessageAlert* Pro? As a starting point we compared the abstract views and found no differences. We therefore know that the advanced version still supports SOAP and HTTP. By examining the transport binding mechanism of WSDL and the structure of HTTP and SOAP messages, we noticed that certain types of changes in WSDL documents require modifications in the client side. We next discuss these conditions, separately considering SOAP and HTTP aspects.

SOAP binding defines how SOAP messages are to be carried within or on top of another protocol. SOAP message can e.g. be embedded in HTTP requests or responses. In some cases, changes in WSDL documents require corresponding modifications in HTTP requests or responses. In this paper, we only discuss the modifications of HTTP requests. We noticed that the client side should be modified in following cases:

1.1 HTTP request line should be updated if differences exist in attributes *location* of *soap:address* elements.

1.2 HTTP line *SOAPAction* should be updated if differences exist in attribute *soapAction* of *soap:operation* elements. SOAPAction field can be used to indicate the intent of the SOAP HTTP request. In SOAP 1.1, SOAPAction field is mandatory but in version 1.2 this feature was made optional.

1.2 Elements in the SOAP body should be updated if differences exist in attributes *type* or *element* of *wsdl:part* elements.

1.4 Wrappers of elements in SOAP body should be updated if differences exist in the attributes *style* of *soap:operation* elements, for instance, the value of attribute style is changed from "document" to "rpc".

We further noticed that HTTP request line, in turn, should be changed in the following situations:

2.1 Differences exist in attributes *location* of *http:address* elements. For example, the value of the location attribute for InstantMessageAlert is "http://www.bindingpoint.com/ws/imalert/imalert.asmx", while for InstantMessageAlert Pro it is "http://www.bindingpoint.com/ws/imalertpro/imalertpro.asmx". Therefore, when the client uses "HTTP GET" to request for sending a message to InstantMessageAlert Pro, the request line should be changed from *"GET /ws/imalert/imalert.asmx/..."* to *"GET /ws/imalertpro/imalertpro.asmx/..."*.

2.2 Differences exist in values of attributes *verb* of *http:binding* elements. For instance, the value of verb is changed from "POST" to "GET".

2.3 Differences exist in values of attributes *location* of *http:operation* elements.

2.4 Differences exist in attributes *type* or *element* of *wsdl:part* elements.

As mentioned above, based on the abstract view comparison, we know that the basic WSDL document structures are same. The differences are found, however, when comparing the views. They exist in attributes of elements wsdl:port, soap:address, wsdl:portType, wsdl:service etc. After exploring the differences we found out that only few of the differences meet some of the conditions above. The conditions met are 1.1 and 2.1. That is to say, client side just needs to modify pieces of code where HTTP request line in either SOAP or HTTP messages is generated.

5 Related Work

Web service evolution has not been intensively researched, which can be due to the fact that large-scale Web service deployment is not yet very common. Managing the evolution of small-scale Web services is possible without specific tool support.

The evolution and versioning of Web services has been addressed by Wilde in [22]. Wilde proposes an approach to both deal with extensions to Web service vocabulary and describe the semantics the extensions. The latter one enables forward compatibility of Web services: older versions of the Web service in question can use the semantics descriptions to dynamically adopt to a later version of the service. The semantics of the extensions are described using a declarative language. While the problem domain comes close to the one we discuss in the second case study (Section 3.2), our approaches differ considerably. We do not aim at describing the semantics of changes in a case of a particular Web service. Instead, we provide a general purpose visualization technique that helps the user to easily identify the changes, based on which she can conclude the sources and semantics of them. We provide support for both comparing the logical differences and the detailed differences between two WSDL instances.

We are not aware of approaches specifically targeted to visualizing the evolutional aspects of Web service descriptions. However, various approaches and tools for visualizing aspects concerning software evolution have been presented. Lanza introduces a concept evolution matrix, which illustrates the evolution of classes in an object-oriented software system [10] based on selected software product metrics calculated for the subject system. By analyzing the evolution matrix, the user can conclude various aspects concerning the size of the system, added and removed classes, and overall growth and stagnation phases during the evolution of the system. Based on case studies, Lanza further identifies various ways the classes evolve over their lifetime. As Lanza, we also detect the removal and addition of classes. In addition, we detect changes in associations. However, we do not aim at visualizing longer change or stagnation trends. Gall et al. present an approach in which a three-dimensional visualization technique is used to support the analysis of the evolution of large software systems at abstract, subsystem level [5]. The 3D visualization used allows showing both system structure and evolution history in the same view. Our approach differs from these approaches e.g. by the visualization technique used; the set operations are applied to UML class diagrams, which may represent either the abstract or low-level views of service descriptions. However, the underlying approach can also be used for comparing subsystem-level models. The visualization

technique used depends on the metamodel selected, but so far we have limited ourselves to the UML notation. In this paper we have only discussed the pair-wise comparisons but the set operations have been designed to also allow the comparison of several models, which in turn could be used to support visualization of evolution histories.

From the viewpoint of aspect-oriented design, the operations can be seen as a restricted mechanism for composing and decomposing models, each describing a single concern or subject. Clarke [4] addresses merging of models through composition semantics, which also play a key role in the Theme/UML approach [2]. The approach is based on extending the UML metamodel and deriving composition relationships between individual subjects. In contrast, the set operations rely solely on the UML standard itself and are implemented and integrated on an existing UML modeling environment. As pointed out by Clarke ([3], pp. 58-60), several development methodologies (e.g. Catalysis by D'Souza [18]) provide different integration approaches for merging models.

Similar UML-based techniques have been presented by Ohst et al. [12] and by Porres and Alanen [16]. The former discuss visualizing the differences between two UML diagram versions, while the latter formalize how to calculate the union and difference of UML models. They both assume that there exists unique repository identifiers and that the different models share a common ancestor. Further, they rely on using proprietary UML tools.

6 Summary

In this paper we have discussed an approach to analyze and compare WSDL descriptions at UML level. The WSDL documents are first transformed to UML class diagrams. Since these low-level views, in which each WSDL element is represented as a UML class, are typically large and contain redundant information, they are next abstracted to more compact UML class diagrams that visualize the logical structures of the WSDL descriptions. UML set operation can then be used to compare either the low-level or abstract views. The result of an application of the set operations is also given as a UML class diagram; differences between UML models are distinguished by different colors. Studying the UML diagrams, users are able to easily conclude differences existing in WSDL documents.

If logical differences can be found by applying the set operations to the abstract views, the sources for these differences can be examined by comparing the low-level views. In some cases the abstract views do not necessarily differ but detailed differences may still occur and they can be found by comparing low-level views.

The key for set operations is the correspondence relationship that can be defined differently for diverse purposes and focus points. The coloring used in the result model indicates the differences found according to the chose focus points. In the case studies presented in this paper, structural relationships (i.e., differences in containment relationships of elements), element naming, namespace definitions, and possible extensions used were chosen as such focus points.

In the first case study we compared popular toolkits supporting the construction of Web services by implementing the same example Web service using four different toolkits and by comparing the generated WSDL documents. A summary of the differences found was presented. Since these toolkits indeed vary in how the WSDL documents are generated, tool support for comparing such differences is useful.

In our second case study we analyzed and compared WSDL documents composed for two versions of InstantMessageAlert Web service by BindingPoint. Our approach provides useful and applicable support for version control and maintenance of Web service descriptions. Especially if WSDL descriptions are generated automatically from service interfaces, the evolution of the interface may require a new version of the WSDL description to be generated. This, in turn, may require changes in the client sides. Therefore, from the clients' point of view it is important to understand and analyze the changes in WSDL descriptions. Our approach provides such support by allowing the WSDL descriptions to be visualized and compared at different abstraction levels in UML.

Parts of the approach presented in this paper are based on previous work by the authors. The methods for transforming WSDL descriptions to UML class diagrams and abstracting them have been presented in [8]. That paper further introduces various structural rules related to WSDL descriptions presented as profiles, one of them being the WSDL profile. The UML set operations, in turn, have been presented in [17]. In this paper we combined these works to form an approach to analyze and compare WSDL descriptions, present two different case studies in which the approach has been used, and present a detailed analysis of the results of the case studies.

7 Acknowledgements

This research has been financially supported by TEKES, Nokia, Plenware Group, and Solita. The authors would like to thank Jan van der Ven for implementing set operations and Jani Airaksinen for all the valuable technical support. The authors would also like to thank BindingPoint for collaboration as well as Kai Koskimies and Tommi Mikkonen for their valuable comments.References.

References

1. J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä, "xUMLi: Towards a Tool-independent UML Processing Platform", In: K. Østerbye (Ed.), Proc. of the 10th NWPER Workshop, IT University of Copenhagen, Denmark, 2002, pp. 1-15.
2. E. Baniassad, S. Clarke, Theme: An Approach for Aspect-Oriented Analysis and Design, In Proc. ICSE'04, Edinburgh, Scotland, May 2004.
3. S. Clarke, Composition of Object-Oriented Software Design Models, PhD Thesis, Dublin City University, 2001.

4. S. Clarke, Extending standard UML with model composition semantics, Science of Computer Programming, Volume 44, Issue 1, pp. 71-100. Elsevier Science, July 2002.
5. H. Gall, M. Jazayeri, and C. Riva, Visualizing software release histories: The use of color and third dimension, In Proc of ICSM'99, IEEE Computer Society, 1999.
6. IBM, IBM WebSpehere SDK for Web Services,
http://www-136.ibm.com/developerworks/webservices, 2004.
7. J. Jiang, J. Lipponen, P. Selonen, and T. Systä, UML-level analysis and comparison of Web service descriptions, In Prof. of CSMR 2005, pp. 236-240.
8. J. Jiang and T. Systä, UML-Based Modeling and Validity Checking of Web Service Descriptions, a manuscript.
9. R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf, A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering, In Proc. of WCRE 2002, 2002, pp. 22-33.
10. M. Lanza, The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques, In Proc. of IWPSE'01, IEEE Computer Society, 2001, pp. 28-33.
11. Microsoft, .Net Framework,
http://msdn.microsoft.com/webservices/, 2004
12. D. Ohst, M. Welle, and U. Kelter, Differences between versions of UML diagrams, In Proc. of ESEC'03, 2003, pp. 227-236.
13. Oracle, Oracle Application Server for J2EE, http://www.oracle.com/technology/tech/webservices/, 2004.
14. J. Peltonen, Visual Scripting for UML-based Tools, In Proc. of ICSSEA 2000, Paris, France, December, 2003.
15. J. Peltonen and P. Selonen, An Approach and a Platfrom for Building UML Processing Tools, In Proc. of WoDiSee'04 workshop of ICSE'04, 2004, pp. 51-57.
16. I. Porres, and M. Alanen, Difference and Union of Models, In Proc. of UML 2003, San Fransisco, USA, 2003, pp. 2-17.
17. P. Selonen, Set Operations for the Unified Modeling Language, In P. Kilpeläinen and N. Päivinen (eds.), In Proc. of SPLST'03, Kuopio, Finland, June, 2003, pp. 70-81.
18. D. D'Souza, and A.C. Wills, Objects, Components and Frameworks with UML, The Catalysis Approach, Addison-Wesley, 1998.
19. Sun Microsystems Inc., Java Web Service Developer Pack, http://developers.sun.com/techtopics/webservices/, 2004.
20. The Object Management Group: Unified Modeling Language Specification, version 1.5 (formal/03-03-01), March, 2003. On-line at http://www.omg.org/uml/
21. J. van der Ven, An Implementation of Set Operations on UML Diagrams. Master's thesis, Rijksuniversiteit Groningen, Instituut voor Wiskunde en Informatica, 2004.
22. E. Wilde, Semantically Extensible Schemas for Web Service Evolution", In Proc. of ECOWS'04, Erfurt, Germany, September 2004.
23. World Wide Web Consortium, http://www.w3.org/, 2004.
24. World Wide Web Consortium, Web Services Description Language (WSDL), http://www.w3.org/, 2004.
25. World Wide Web Consortium, Simple Object Access Protocol (SOAP), http://www.w3.org, 2004.