Proceedings of NWUML’2004
2nd Nordic Workshop on the Unified Modeling Language
Proceedings of

NWUML'2004

2nd Nordic Workshop on the Unified Modeling Language

August 19-20, 2004, Turku, Finland

Editors:

Kai Koskimies
Ludwik Kuzniarz
Johan Lilius
Ivan Porres

TUCS General Publication
No 35, August 2004
Preface

Welcome to the 2nd Nordic Workshop on the Unified Modeling Language held in Turku, Finland, August 19-20, 2004. The objective of the NWUML workshops is to bring together researchers and Ph.D. students in the fields of software and system modeling, including new and standard UML-based languages, model-based development methods and modeling tools.

The international UML conference is the top scientific venue to present new research on the topics of UML and modeling. However, the conference is organized alternatively in America and Europe. Besides this conference, we consider that there is a need for a workshop located in the Nordic countries to foster discussion and establish a research community in UML and related areas. This year, the workshop is colocated with the 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER’2004. The topics of software modeling and software development tools are certainly related and we hope that this will increase the possibilities to establish new contacts between researches of these two communities.

The first Nordic UML Workshop was organized in Rönneby, Sweden in 2003. In this second edition, 14 articles will be presented covering important aspects of UML and modeling technology such as model semantics, model transformations, verification and analysis of models and modeling profiles for different application domains. We consider that these articles represent an excellent sample of the ongoing research on these topics in the Nordic countries and Europe.

We hope that you enjoy the workshop!

Turku, August 2004

Ivan Porres
Program Committee
Görel Hedin, Lund Institute of Technology, Sweden
Kai Koskimies, Tampere University of Technology, Finland
Johan Lilius, Åbo Akademi University, Finland
Eila Niemelä, VTT, Finland
Jan Pettersen Nytun, Agder University College, Norway
Ian Oliver, Nokia Research Center, Finland
Jari Peltonen, Tampere University of Technology, Finland
Juha-Pekka Tolvanen, MetaCase, Finland
Andreas Prinz, Agder University College, Norway
Mika Siikarla, Tampere University of Technology, Finland
Mirosław Staron, Blekinge Institute of Technology, Sweden

Workshop Organization
Kai Koskimies, Tampere University of Technology, Finland
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden
Johan Lilius, Åbo Akademi University, Finland
Ivan Porres, Åbo Akademi University, Finland
Contents

1  Representation of Levels and Instantiation in a Metamodelling Environment  
   Jan Pettersen Nyten, Andreas Prinz, Andreas Kunert  
   1

2  Semantics of Structured Nodes in UML 2.0 Activities  
   Harald Störrle  
   19

3  UML-Based Testing: A Case Study  
   Marko Jäntti, Tanja Toroi  
   33

4  Modeling Concern-Based Testing with UML Testing Profile  
   Mika Maununmaa, Mika Katara  
   45

5  HAZOP Analysis of UML-Based Software Architecture Descriptions of  
   Safety-Critical Systems  
   Klaus Marius Hansen, Lisa Wells, Thomas Maier  
   59

6  Factors Determining Effective Realization of MDA in Industry  
   Miroslaw Staron, Ludwik Kuzniarz, Ludwik Wallin  
   79

7  The Coral Modelling Framework  
   Marcus Alanen, Ivan Porres  
   93

8  Putting Concern-Oriented Modeling into Practice  
   Stéphane Bonnet, Raphaël Marvie, Jean-Marc Geib  
   99

9  Metamodels and architecture of an automatic code generator  
   Ignacio García-Rodríguez de Guzmán, Macario Polo,  
   Mario Piattini  
   115

10 Good practices for creating correct, clear and efficient OCL  
    specifications  
    Dan Chiorean, Dyan Corutiu, Maria Bortes, Ioana Chiorean  
    127

11 Performance Modeling and Visualization with UML  
    Juha Gustafsson  
    143

12 Towards the Unification of Patterns and Profiles in UML  
    Petri Selonen, Kai Koskimies, Tommi Mikkonen  
    151

13 An Approach for an UML Profile for Software Development Process  
    Modeling  
    Ralf Buschermöhle, Wilhelm Hasselbring  
    167

14 UML Profile for Analysis and Design of Jakarta Struts Framework Based  
    Web Applications  
    Piotr Karwaczyński, Łukasz Maciejewski  
    185
Representation of Levels and Instantiation in a Metamodelling Environment

Jan Pettersen Nytun¹,², Andreas Prinz¹, and Andreas Kunert³

¹ Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
jan.p.nytun@hia.no, andreas.prinz@hia.no

² Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway

³ Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
kunert@informatik.hu-berlin.de

Abstract. In the scope of meta-modelling it is important to consider descriptions sometimes as a model and sometimes as a metamodel, e.g. the UML metamodel which is a metamodel for UML and at the same time a model in terms of MOF. For this handling to be easy, this article describes a meta-level representation that includes both aspects. It covers the most important relations within the MOF framework starting with plain objects and relations. A prominent role plays the instantiation, which is the only connection between levels. The article explains how instantiation is represented and which kind of constraints are related to it. This way it is possible to bring metamodelling to its basic semantics.

1 Introduction

There is a trend towards the use of languages that are tailored for special problem domains and also towards integration of different languages; in an OMG context this can be done by using the extension mechanisms of UML, definitions of UML profiles, and also definition of new MOF [6] metamodels. Tools that support definition and application of this type of languages are largely missing. Some of the goals of the SMILE (Semantic Metamodel-based Integrated Language Environment) project are to design and implement a framework that can be used to do metamodelling, integration of different languages and even support for the definition of new meta-metamodels different from MOF.

Our SMILE project targets all the levels of the OMG four-layer metamodel architecture, and we are proposing a basic representation to be used on all levels. We call our representation FORM after its main construct Form.¹ The term carrier from the telecommunication field has many similarities with our representation since all types of objects and classes are to be carried (represented) with it.

All levels, also the lowest level, will be covered in our basic representation; this lays the ground for executable UML with "metadata facilities" and it also makes it possible to specify how two neighboring levels fit together as described later.

As an example we discuss how FORM can be used to "carry" the different levels of the metamodelling architecture of UML. Afterwards we consider how instantiation relates to FORM.

Except for instantiation, semantics is not considered, but will be dealt with at a later stage. Even without full support for semantics the framework has interesting applications

¹ We also consider FORM to mean "Fantastic Organized Representation of Models".
like analyzing static data (e.g. a snapshot of a running system) and check if an instance is consistent with a given model. To do these types of applications we would also need OCL. Relating OCL and XMI with FORM is briefly discussed.

This paper is structured as follows. Section 2 gives an introduction to metamodelling and representation of metadata, followed by an overview of related work in Section 3. We introduce our FORM representation in Section 4. In Section 5 we discuss why this simplified representation is able to cover all the things already within XMI, hence covering all that is necessary for models. We define the semantics of instantiation within Section 6 and offer some conclusions and directions for further work in Section 7.

2 Metamodelling

Today the metamodelling approach is a common way of organizing models, a way that involves descriptions on levels placed on top of each other. The concepts of one level have corresponding descriptions on a next level (metalevel, level above). Stated differently, a level is a model and the level below is an instance of this model. In relation to object-oriented programming the lowest level contains the objects of a running system, while the classes reside on the next lowest level. Traditionally, the BNF notation has been used to describe a programming language, this would be the next level. The top level would be a definition of BNF\(^2\). These 4 levels correspond to the 4 levels of the four-layer metamodel architecture of OMG, but here visual UML models are used instead of BNF. See Fig. 1 for an overview.

![Fig. 1. The four-layer metamodel architecture](image)

The advocated architecture is based on strict metamodelling which means that all elements on one level are instantiated from the level directly above. It seems natural to view the instantiation logic as operating with 3 levels, e.g. you have a description of what a class is, you have a class (e.g. class Person) and you have an object. The definition of the class concept must of course be done with the help of yet another level.

\(^2\) Please note that we do not need a level above BNF, because BNF can describe itself.
MOF and UML do both offer support for object-oriented concepts and the core parts of MOF and UML are structurally equivalent. Since MOF is used to define itself, the level above MOF (M4) can be seen as MOF once more; one can imagine an infinite number of MOF levels - we have infinite regression. MOF is defined by self-referencing, "ending up with" class `Class` which is an instance of itself.

The syntax of UML has been described in a notation independent way; this abstract syntax defines the elements of UML and how they relate to each other. There is also an agreement on the concrete syntax of UML. The concrete syntax of a simple class is a rectangle with the class name inside the top compartment and optional compartments for attributes and operations. An object is described as a rectangle with a top compartment containing an optional object name and then after a colon its class, the complete text is underlined (e.g. `Bob:Person`); an object can also have a second compartment for slots with values (attribute values).

![Diagram](image)

**Fig. 2.** Different representations of the same structure.

You can only display modelling elements by using concrete representation; the abstract syntax is usually defined with class diagrams together with OCL [4] constraints. An instance of the UML metamodel can be shown as an object diagram or as a class diagram; the class diagram being a visual interpretation of the underlying object-graph. In a sense the class diagram notation is using syntactic sugar, e.g. an attribute is shown inside a compartment and not as an instance of `Property`. A class diagram is a natural choice when you see a level from the level below, while an object diagram shows how the level has been instantiated from the level above.

A part of the MOF (abstract syntax) is shown in Fig. 2(a) on level M3, it deals with the structure of classes and states that a class can own attributes (properties). Fig. 2(a)
shows how the abstract syntax has been instantiated to form the UML concept Class with attributes name and isAbstract on level M2; the class Class on level M2 is an instance of class Class of level M3 and the attributes are instances of class Property. Similarly, the class Person on level M1 is an instance of the UML concept of Class, and on level M0 there is even an instance of the class Person. Fig. 2(b) show the same with clabjects. In [2, 1] an entity with class and object nature is called a clabject, in fact all classes are clabjects. The clabject-notation allows you to see the attributes and the slots of a class, this can be shown in separate compartments as done in the figure. All the classes of Fig. 2(a) are represented as clabjects in Fig. 2(b); Fig. 2(b) is more explicit: for instance the name attribute of class Class on level M3 has been instantiated to a slot with value "Class" on level M2, class Class on level M2 can be instantiated since slot isAbstract has value false, the attribute name is an instance of Property.

Fig. 3. Representing the same underlying structure as Fig. 2, but more explicit.

We can make this representation even more explicit by also showing the relations between the classes and their attributes as in Fig. 3. Here we have shown all attributes as separate Property objects being associated with their defining class. Please note that for full completeness we also would have to insert a definition of the association between Class and Property on level M3.

3 Related Work

Metamodelling has been discussed for a long time, to mention a few articles: [2] describes the unification of the class and object facets (clabject); the same article also presents

The definition would be like: Class:Class (already on M2) linked to :Property linked to :Association linked to :Property linked to Property:Class (already on M2). Both the links on M2 could now have an instanceOf relation to :Association on level M3. For the sake of understandability we have omitted these parts on the figure.
an elegant enhancement of the instantiation mechanism to allow definitions to transcend multiple levels; [3] aims at improving the metamodelling architecture of UML.

There are several metamodelling repositories based on the Java Metadata Interface (JMI) [9] specification. JMI is based on MOF and makes it possible to create and access metadata; this can be done at design time or runtime by using the reflective JMI API or via a set of generated metamodel-specific APIs. A tool like Metadata Repository (MDR) [11] can import a model represented in XMI and automatically produce JMI interfaces for accessing the metadata and also automatically provide implementation of the JMI interfaces.

The Eclipse platform [12] is designed for building integrated development environments (IDEs). It provides already a working IDE for Java development and there are plugins for drawing UML diagrams. The Eclipse Modeling Framework (EMF) [13] includes a metamodel (Ecore) that is different from MOF; there is an EMF-based implementation of the UML 2.0 [14] metamodel. The functionality of EMF is quite similar to MDR.

All the mentioned Java-based approaches are using the class/interface concept of Java to implement the metaclasses, and the semantics are not handled separately as we plan to do.

The Coral Modelling Framework [8] is not based on Java but on the Python programming language; the meta-metamodel (MOF) is hard coded but different metamodels can be installed automatically; there is currently no support for the lowest level. The framework support some advanced features, like transaction control of model updates and scripting of queries in Python. This approach is quite similar to the Java approaches, but here the class mechanism of Python has been used.

One major source of inspiration has been the UML and MOF specifications; inspecting these documents made it clear that a level can be represented as a graph consisting of connected objects; the concepts related to describing a class must then also be represented by objects, e.g. an attribute is an instance of Property linked to another object representing the class. Also by examining the possible instantiations described (e.g. an association is to be instantiated as a link), we got a picture of how the instanceOf-relation applies. We are designing a metamodelling framework, and are concerned about how two single levels can be connected in a correct way. Those types of questions are not answered by a repository-based approach. In our approach we separate out the representation, and then allow attaching of different semantics.

4 FORM: Our Basic Representation

As our aim is to represent models the same way as metamodels, we propose in this section a basic representation for models called FORM, which can be used independently of the level of the models. Our proposed representation is based on the following observation: *Every part of a level in the metamodel structure is an instance of something at the level above.*

Generally an instance has been instantiated from the level above, but the top level is special: everything at this level can be seen as an instance of something residing on this same level. This can be seen as using the same level for describing itself.

The metamodelling environment can be restricted to objects; in a sense ”object” is the lowest common denominator of all levels. Since our metamodelling approach is object-oriented, we need to represent the following information:

- Objects.
- Slots owned by objects.
- Slot values and their type.
- Links between objects.
- Every object knows its class.
- Every slot knows its attribute.
- Every link knows its association.

One metalevel is an *understanding* (meaning, semantics) of how the level below can look like; this understanding is necessarily coded into the structure of the level. To make such a structure one needs a carrier - a substance that can be formed into a coded understanding. We are used to papers and pencil, our FORM representation on the other hand is meant for computers.

![Fig. 4. FORM Context](image)

As we can see in Fig.4, FORM is providing an abstract representation of models (e.g. UML models). We can describe the constituents of this representation as done in Fig. 5, where we define physical building blocks that can be put together to form complex structures, much like how atoms can be put together to form molecules.

![Fig. 5. FORM - the basic representation](image)
The metaphor above used the term physical which in this context can be seen as the same as concrete, one can choose to see the computer objects as existing physical objects [10]. "Abstract information" must be coded in concrete representation; the coded information is forming a pattern (structure) that might be interpreted by humans or machines. A computer can interpret in the sense that it can transform and operate on the structure.

It is of course problematic to talk about abstract representations and even to handle them. Therefore we also need a concrete (physical) representation. For our implementation this can turn out to be computational objects, e.g. Java objects. For the sake of this article, we do also need a representation that can be put onto paper. We define such a representation in section 4.2.

4.1 Definition of FORM

Fig. 5 shows a model of our basic representation; where we have Form, Slot, Value and Link as four special types of instances. Objects will be represented as instances of Form, links as instances of Link and slots as instances of Slot.

Form will be involved in the representation of both objects and classes; it seems that Form is a good name since it has the following two meanings: "the shape and structure of something" and "a mold for shaping something". We also consider a value to be an instance, this is further discussed in section 6.

Fig. 6. Viewing the levels as components

An instance of class Symbol (typically the name of what it represents), will have a link to the Form-instance it symbolizes. The mentioned Form-instance, together with connected slots and possibly some other linked Form-instance, can be seen as an interpretation of the symbol. Instead of letting an instance be linked directly to what it is an instance of, we have an interface of Symbols between levels. This allows us to handle
levels separately from their adjoining levels and only bringing them together when necessary. It is easy to relate the model to a metalevel border: the interpretation of a symbol is on the upper side of the border and the "instances of the symbol" are on the lower side of the border. The symbols of the instantiable forms of one metalevel constitute the interface towards the level below.

Fig. 6 shows how the levels can be seen as separate components; each component provides an interface which can be thought as the symbols on the lower border of the level it represents. On the other hand, each component is requiring an interface which can be seen as the upper border of the level it represents. The MOF-component is special since its required interface is the same as the interface it provides. Also level $M_0$ (User Data) is different since it provides only an empty interface.

4.2 The FORM Notation

In order to be able to visualize the (abstract) FORM representation, we need some notation. Of course it would be possible to use some kind of UML notation, because UML includes all aspects that are necessary here. However, as our notation is used for representing models and metamodels, it usually leads to confusion if we use a known notation. Therefore we use the symbols shown in Fig. 7 for visualizing the elements of our abstract representation. We visualize a reference as a line connecting the two involved entities (arrows are used to indicate the direction of the reference; if there are two references going opposite ways we omit the arrow-head). The $\lambda$ inside Symbol instance is to be replaced with the actual name of the Symbol; the $\lambda$ inside Value instance is to be replaced with the value (e.g. 5).

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Visual Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol instance</td>
<td><img src="image" alt="Symbol instance" /></td>
</tr>
<tr>
<td>Form instance</td>
<td><img src="image" alt="Form instance" /></td>
</tr>
<tr>
<td>Slot instance</td>
<td><img src="image" alt="Slot instance" /></td>
</tr>
<tr>
<td>Link instance</td>
<td><img src="image" alt="Link instance" /></td>
</tr>
<tr>
<td>Value instance</td>
<td><img src="image" alt="Value instance" /></td>
</tr>
<tr>
<td>Connection (Reference)</td>
<td>![Connection (Reference)]</td>
</tr>
</tbody>
</table>

Fig. 7. The visual symbols of the FORM-notation.

In Fig. 8 our mechanism is used with the example of Fig. 2. As we can see, a (UML) class is represented as a Form, a slot as a Slot, and so on. It is obvious that this again is just an even more explicit representation, the structure represented is still the same\(^4\). From Fig. 8 it can be clearly seen that our representation is capable of representing all levels in a uniform way using only the few notational elements given in Fig. 7.

\(^4\) Please note that the simplifications done in the earlier figures are inherited and also that some symbols are duplicated.
4.3 InstanceSpecification in MOF 2.0 and UML 2.0

The object notation with the instanceOf-relations introduced in figure 2 is not defined in [6, 5, 7]. The concrete syntax of InstanceSpecification is quite similar, but the purpose of InstanceSpecification is not to relate an object to its class at the level above. The UML specification [5] states: "An instance specification is a model element that represents an instance in a modeled system... An instance of an InstanceSpecification is defined in a model at the same level as the model elements that it illustrates".

An extract of the UML metamodel is shown in figure 9. As we can see, there will be no reference to the level above when it comes to instances of InstanceSpecification or any of the other classes. Our conclusion is that InstanceSpecification does not fill our needs, we must have references to the level above.

5 XMI

In this section we want to discuss if the FORM representation is sufficient to represent models stored using XMI and describe how a production of XMI files starting from models represented by FORM looks like. In SMILE we plan to use XMI as the exchange format. The main advantage is that XMI is a widely accepted standard, so we are able to exchange our models with modelling tools from other vendors. This is especially important in the project startup phase, where we can use XMI documents created by other tools as test samples.

XMI is a standard for exchanging and storing models using XML. The current version is 2.0 released by the OMG on May 2, 2003. The standard defines a number of production
rules which are used to read/store models in/from XML files. The main production rules are:

- production of XML Schemas starting from an object model
- production of XML documents compatible with XML Schemas
- production of XML documents starting directly from objects
- reverse engineering from XML to an object model

The first three methods are used for saving model data while the last one is needed for restoring. There are at least two ways of getting an XML file out of a model. The longer one is:

- Create an XML Schema which represents the metamodel (in older version of the XMI standard Document Type Definitions (DTD) were used instead of XML Schema).
- Create an XMI File according to the XML Schema containing the model.

The shorter way consists only of one step:

- Directly create an XMI File representing the model.

As you can see, the creation of XML Schemas representing the meta-model is optional. It is recommended nonetheless because it allows you to validate your model (stored in an XMI file) against the meta-model without having to decode the XMI file first. Moreover it gives you a possibility to define unique representations of certain model elements when the XMI standard allows multiple ways of encoding them.

However, it is still possible to store invalid models (according to the meta-model) in a valid XML file (according to the XML Schema produced out of the same meta-model) due to some semantic constraints of the meta-model which cannot be expressed in the XML Schema. Thus XML Schemas of meta-models provide you necessary, but not sufficient criteria for validating models stored in XMI.

If we have a model stored using the FORM representation, we can create an XMI file using the following rules:

Every model element stored as Form in FORM becomes an XML element in XMI. This XML element is named like the Symbol the Form is described by, and has an attribute xml:id whose value is unique among the saved forms (we will use xmi as the name for the namespace http://www.omg.org/XMI in the following examples). A Form implementing a computer (defined in the metamodel) would be written like the
Attributes, stored in Values and referenced via Slots in FORM, can be represented in XMI in two different ways. They can either become XML attributes of the corresponding XML element (the XML element of the Form the attribute belongs to) or XML elements that belong to the XML element of the owner. In the first case, the name of the Symbol describing the Slot is represented by the name of the XML attribute. In the second case, it becomes the name of the XML element.

Considering the computer mentioned in the example above had an attribute called os (which is stored using a Slot implementing the Symbol os). The XMI representation of two computers running two operating systems could be represented by the following XMI fragment (both possibilities are shown):

```xml
<computer xmi:id="c01" os="linux"/>
<computer xmi:id="c02">
  <os>unix</os>
</computer>
```

Associations are represented by Links in FORM. Like attributes they can also be stored using either XML attributes or XML elements. An XML attribute is named like the association (i.e. the Symbol it is described by) and carries the xmi:id of the referenced element(s) as value. An XML element is named in the same way and it contains the referenced element(s) in the href attribute. If we add an association user from computers to persons in our example, we can represent two computers which are both used by the same two persons like that:

```xml
<person xmi:id="p01"/>
<person xmi:id="p02"/>
<computer xmi:id="c01" user="p01 p02"/>
<computer xmi:id="c02">
  <user href="#p01"/>
  <user href="#p02"/>
</computer>
```

Apart from some minor details the transformation process from a FORM model to an according XMI representation is rather simple. However, the process of restoring XMI data is much more complicated.

It is obvious that we are able to import XMI files formerly exported by ourselves. It is harder to import XMI files from other vendors.

The biggest problem is that XMI allows different representations of the same model construct. Moreover there are some XMI constructs that we do not plan to use, but we have to expect tools from other vendors using them. However, we can represent all these constructs using FORM.

5 c01 is just an automatically generated identity of this entity. It is not needed in this small example.
The only exception to this statement we know are the so-called vendor extensions. These are extension elements in an XMI file that can be used to store additional data which is only important for the tool which exports (and reimports) the XMI file (e.g. graphical modelling tools can use vendor extensions to store the screen positions of model elements).

The standard behaviour of XMI applications is to ignore all unknown vendor extensions. In SMILE we plan to deal with vendor extensions the same way. This is not only unavoidable due to the fact that the data in vendor extensions is unstandardized, but also acceptable since vendor extensions should not contain model data.

To summarize this section we can say that we are able to convert every model from a FORM representation to an XMI representation and vice versa. This means that FORM is sufficient to store any kind of models.

6 Instantiation

The most interesting action with metamodelling is instantiation, meaning creating a level using the information on the next higher level. However, metamodels are declarations; the act of instantiating a metamodel is not described, and only the set of possible correct results is described\(^6\).

Therefore we will discuss in this section the problem of deciding if two given single levels can be seen as neighboring levels. What are the semantic consequences of connecting two levels and what must the underlying structure support to achieve a “sound” connection?

The discussion will be concerned with two adjacent levels, a lower level (object/instance level) and an upper level (meta level). We consider basic instantiation patterns for all the elements of our FORM representation, i.e.

- Instance, see 6.1
- relation between Symbol and Instance, see 6.1
- different kinds of Instances, see 6.3
- relation between Form and Slot, see 6.2
- relation between Link and Slot, see 6.2
- relation between Slot and Value, see 6.2
- built-in values, see 6.4.

We only consider the symbol interfaces related to the instantiation, i.e. the upper interface of the lower level (client interface) and the lower interface of the upper interface (server interface). In our approach, we do not have any information coded ”into the symbols”. This way the names do not provide semantics.

6.1 Basic Instantiation - Matching Symbols

Any instance on any level has to be related to a defining Form on the level above. The most basic pattern describes this: an Instance on the object level relates to a Form residing on the meta level through matching Symbols of their related interfaces, i.e. level borders. Two Symbols in different borders are considered to be the same Symbol if they have equal names, assuming that the Symbols of each border are unique.

\(^6\) The MOF specification has some instructions on how to go about and also includes a factory-class.
Fig. 10(a) shows two levels that are to be connected; the Symbols are represented as $\lambda_1$ and $\lambda_2$, for a match to occur these Symbols must be the same. Assuming $\lambda_1$ and $\lambda_2$ are equal then the result of matching can be seen in Fig. 10(b).

![Diagram](image)

**Fig. 10.** Connecting a Form to its description.

If some Symbols of the server interface are not merged with Symbols of the client interface, then it simply means that there are no "instances of" these Symbols, which is no problem. If some Symbols of the client interface are not merged with Symbols of the server interface then some instances of the object level are still without definition, which means that the matching was not completely successful - some Symbols are unresolved.

If we assume that this rather uncritical matching of Symbols has created a situation where all Symbols of the client interface are matched, then we have to check that the descriptions on the meta level fits to the instances on the object level. This is described below.

### 6.2 Instantiation of Links

The only possible information that can be given on the meta level so far is structural information in the sense of connections between entities. There are three kinds of connections defined in Fig. 5, ignoring the connections with Symbols which already have been taken care of.

The connections we are talking about are not the links which are instances of Link as part of FORM; the connections we are talking about are the most basic ones and they must be supported by the underlying system (how they are physically represented is up to the underlying system - in Java such links could be implemented as references).

What is the relation between Link and a connection? It is clear that we need some kind of description of an object level connection. This description should be given on the meta level, and according to the possibilities in FORM it has to be a Link instance. However, as Link can only be connected to a Slot, a "complete" link is represented as (form-)slot-link-slot(-form), and connections will be used to bind those entities together (e.g. a connection from a Slot-instance to a Link-instance).

Fig. 11(a) shows two levels that are to be related; the Symbols match and the two levels connected can be seen in Fig. 11(b). On the upper level of Fig. 11(b) we find the pattern: "slot-link-slot" which is what we choose to see as a description of a connection. On the lower level we find the connection between a form and a slot. A connection on the lower level must always have a corresponding instance of the slot-link-slot pattern on the upper level meaning that the slot-link-slot pattern is to be found on the upper level between the descriptions of the two involved entities.
If this condition cannot be met we cannot correctly connect the levels, the description does not fit to the instance. The same logic also applies for a connection from a Slot to a Value, and for a connection from a Link to a Slot.

6.3 Different kinds of Forms

It seems that so far all connections on the lower level are always represented by the same slot-link-slot pattern on the upper level. This is not really sufficient since the three kinds of connection are different, e.g. a slot-link connection is different from a slot-value connection. In order to provide support for this we attach an information to the Forms stating which kind of instance can be related to them. In order to also have this information visible on the interface, we in fact attach it to the corresponding Symbols. In Fig. 12 we have added an enumeration InstanceKind to distinguish the different kinds of instances a Form can create.

The description type (or instance type) cannot be assigned arbitrarily. Take as a complex example the instantiation of Link: To do this a rather huge structure has to be present on the upper level (a complete description of an association). If this structure is not in place we end up with having a description that does not make sense. The support of these things is done by the framework and is not described in this article.

6.4 Handling Values

The only special kind of values we have introduced so far are the Links. However, in real systems it is also necessary to handle primitive data like integers or strings. Looking at UML we find the concept of primitive type, which is called PrimitiveType (or just Primitive); instances of PrimitiveType are primitive types like Integer or String. Instances of these types are values of the domain they represent, e.g. 5 is an instance of Integer. The number 5 is an example of what they call a PrimitiveValue. A primitive type is special since it is implemented or built in by the underlying infrastructure; it is made available for modeling and is accessible at all times and at all levels. Its semantics cannot be found on a level in the metamodel structure - it has no "relevant substructure (i.e. it has no parts)" as they state in [4]. Assume that 5 is used on level M1; 5 could then be seen as an instance of the type Integer which resides on level M2; Integer could then be seen as an instance of PrimitiveType which resides on level M3, and finally PrimitiveType could be seen as an instance of Class.

However, a basic value has already a defined semantics, which can be related to the definition of its (proto)type on the level above. The same is not true for the other instances, they all have an explicit structure as their semantics.
In our approach, however, we do not have the possibility to give special semantics to names - all semantics should be given explicitly or be built-in. Therefore we introduce a special kind of Value for the basic values called BasicValue. We also introduce a special kind of Form for the basic types called BasicType. Both basic types and basic values are not characterized by their internal structure, but they carry their semantics already with them. In a way, they have an "external structure". Handling of external structure is only possible to be built-in, which is what we do here.

The only interesting information about basic values and basic types when it comes to instantiation is their relation: When is a basic value an instance of a basic type? This structural conformance check has two parts. First, the defining form of a basic value has to be a basic type. Second, the basic value should be within the range of the basic type.

The first requirement is easily handled by introducing a new InstanceKind for basic values. The second requirement is handled by introducing a type checking function for basic types, i.e. BasicType::membercheck(Value) → Boolean. The result of all these additions is seen in Fig. 12.

![Fig. 12. FORM with instantiation information](image)

This handling conforms very much to the schema of UML. If we repeat the example with 5: 5 is a BasicValue instance of the type Integer which resides on level M2; Integer is an BasicType instance of PrimitiveType which resides on level M3; PrimitiveType would be a Form which is an instance of Class.

The difference is that we are giving the semantics bottom-up instead of top-down as within UML. In other words, it is completely arbitrary which class the basic types are collected in because we do not attach meaning to names.

### 6.5 Derived Consistency with OCL

The above checks are enough for checking all the semantic things that are defined on the basic model. However, sometimes it is necessary to define some more checks. This is

---

7 As the research on abstract data types has shown it is possible to turn every external structure into an internal structure. However, as was also shown by the ADT community, this is in general very error-prone and should be avoided if there are other possibilities.
done using OCL, i.e. by attaching formulas to the forms which should be checked on the level below.

We take the view that this is still another application of the same instantiation pattern for the OCL parts. This means the OCL formulas are considered to be the templates and are instantiated as values on the level below. In this context the OCL formulas are considered well-formedness rules and the implied semantics is that these values should all be True. However, it is possible to consider any other use of the values generated by the OCL formulas.

We want to give the semantics of the OCL formulas directly on the OCL metamodel, not within the implementation. This way it is possible to consider OCL as just an additional module allowing a way of semantics description. In a similar way also dynamic semantics can be defined as a separate module.

7 Conclusions and Research Directions

In this article, we have described FORM, the abstract syntax of a language for representing levels in a metamodelling environment. An instance of FORM will be an object-graph were objects have slots with values and the objects are linked to each other by a link value (instance of Link). Form is a powerful construct, an instance of Form is always an object, but it can also be used to represent a class, a class for classes and so on.

This is achieved using the very basic structures present in all the models and also within the UML and MOF. We have taken this to the extreme and removed all special cases in order to get a level independent representation.

This way all entities are understood to be created according to some object or pattern on the level above. Similarly, all Forms have the power to create objects on the next level. In this article, we have tried to give a better understanding of the process of instantiation, or the semantics of it.

Using our approach we can represent all the things possible to be represented using XMI, i.e. all models. This representation does not yet include any semantics apart from the instantiation.

We envision to provide ways to attach semantics to the meta model elements making it possible to use this semantics on the next level to describe static and dynamic properties, textual and graphical representations or exchange formats.

References


Semantics of Structured Nodes in UML 2.0 Activities

Harald Störrle
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, GERMANY
stoerrle@informatik.uni-muenchen.de

Abstract. The recent major revision of the UML [21] has introduced significant changes and additions to “the lingua franca of Software Engineering”. Within the UML, Activity Diagrams are particularly prominent, since they are the natural choice when it comes to the modeling of web-services, workflows, and service-oriented architectures. One of the most novel concepts introduced are so called structured nodes (including loops, collection parameters, and streaming). Building on [29–32], this paper explores the meaning of StructuredActivityNodes, as it is called in the metamodel, by defining them mathematically in terms of procedural colored Petri nets.

Keywords: UML 2.0, Activity Diagrams, structured nodes, loops, collection-valued parameters, streaming, procedural Petri-nets

1 Introduction

1.1 Motivation

The modeling of business processes and workflows is an important area in industrial software engineering, and, given that it crucially involves domain-experts which are usually non-programmers, it is one of those areas, where model-driven approaches definitely have a competitive edge over code-driven approaches. As the UML has become the “lingua franca of software engineering” and is the cornerstone of the Model Driven Architecture initiative of the OMG, it is a natural choice for this task. Within the UML, Activity Diagrams are generally considered to be the appropriate notation for modeling business processes, workflows, and system-level behaviors, such as the composition of web-services. Unfortunately, the ActivityGraphs1 of UML 1.5 have certain shortcomings in this respect, one of which is the lack of structuring mechanisms withing ActivityGraphs. The only way to model non-linear control- and data flow is the DecisionNode, that is, a goto, with all its negative repercussions.

There are none of the structuring mechanisms in the UML 1.5 that have replaced gotos in programming languages, like different kinds of loops, properly nested if/then/else or case/otherwise expressions, exceptions, transactions or similar. Consequently, it is quite understandable that some people consider the ActivityGraphs of UML 1.5 as “spaghetti-diagrams”.

The OMG has addressed this problem by adding so called StructuredActivityNodes (“structured nodes”, for simplicity) in the new version of the UML. However, these constructs are totally new, not just in the UML, so there is little to no experience yet, concerning both their concrete and abstract syntax. Also, their semantics is described only in

1 Adapting the convention of the standard, words with unexpected initial Capitals or “CamelCaps” refer to meta-classes.
a very superficial way, without the formal rigor necessary for practical model exchange, or enactment, or verification tools. This paper strives to improve in this respect, providing formal definitions for the various types of Structured Activity Nodes, and exploring the new notions both syntactically and pragmatically.

1.2 Approach

Since the standard stipulates that Activities “use a Petri-like semantics” (cf. [21, p. 292]), it is natural to use Petri nets as the semantic domain. For various reasons, plain PT-nets are not sufficient, however. Building on my previous work, we use Procedure-call Colored Petri-nets.

In [29–32], I have shown how control-flow, procedure calling, data-flow, and exceptions in UML Activity Diagrams can be mapped to higher-order variants of Petri Nets. The question now is, how the remaining constructs—LoopNodes, ConditionalNodes, and ExpansionRegions—can be embedded: are they just syntactic sugaring, or do they require semantic additions? Also, the notion of streaming put forward in the UML obviously raises questions related to concurrency. And, putting all this together: is it possible to stretch the approach presented in [29–32] a bit further to also cover structured nodes?

2 Activity Diagrams

A detailed discussion of the concrete and abstract syntax of UML 2.0 Activities, the semantic domains of procedural and colored Petri-nets, respectively, and the semantic mapping of control- and data-flow, and exceptions of Activities is found in [29], [30], and [31, 32]. For brevity, we only give a short summary of the intuition here.

Compared to UML 1.5, the concrete syntax of Activity Diagrams has remained mostly the same, but the abstract syntax and semantics have changed drastically (see Figure 3). While in UML 1.5, Activity Diagrams have been defined as a kind of State Machine Diagrams (ActivityGraph used to be a subclass of StateMachine in the Metamodel), there is now no such connection between them: “Activities are redesigned to use a Petri-like semantic” (cf. [21, p. 292]). The intuition of the semantic mapping is presented in the first six compartments of Figure 2.

The whole area of Activity Diagrams has been structured into a graph of packages (see Figure 1) of increasing expressive power. This paper deals with the constructs of the package CompleteStructuredActivities.

For elementary Activities, the mapping to Petri-nets is rather simple. Intuitively, Actions that are ExecutableNodes become net transitions, ControlNodes become net places or small net fragments, and ActivityEdges become net arcs, possibly with auxiliary transitions or places. For data-flow, the mapping is similarly easy, but requires colored Petri-nets as the semantic domain to cover data-types, guards, and arc-inscriptions. As Actions may call Activities, transitions should be able to call nets like procedures. Obviously, this goes beyond traditional P/T-nets, so that [29] resorts to procedural Petri-nets (first described in [18]). We require that each Activity is represented by a separate boxed and named Activity Diagram similar to UML 2.0 Interaction Diagrams (cf. Figure 6 and [33, 28]), each of which may then be transformed separately into a plain Petri-net. These individual nets are held together by a refinement relationship exploited at run-time, i.e., when a family of Petri-net is executed.

For the intuition of Activities without StructuredActivityNodes see Figure 2. There, each field shows the intuition of the translation for one area of Activities. Due to the
Fig. 1. Levels of expressiveness in Activity Diagrams.

restricted space in this paper, the mapping is given only intuitively, with Activity Diagram fragments (left column) to Petri-net fragments (right column). The details are defined in [29, 30]. Exceptions are left out (see [31, 32]).

Fig. 2. The intuition of the semantic mapping for Activities. Actions that call Activities are represented as Petri-net transitions with a double outline. They are translated into refined transitions of a procedural Petri-net.
3 Abstract syntax and semantic domain

In UML, the abstract syntax is defined by the metamodel. Figure 3 shows a small portion of the metamodel concerned with StructuredActivityNodes. Obviously, the metaclass StructuredActivityNodes is a subclass of Action. Every StructuredActivityNode may contain ActivityNodes and ActivityEdges. StructuredActivityNode has three subclasses, ConditionalNode, LoopNode, and ExpansionRegion, which are examined in sections 4, 5, and 6 in turn.

![Diagram of the UML 2.0 metamodel as far as ExpansionNodes are concerned (simplified).](image)

The semantic domain consists in a rather arcane Petri-net dialect, so it is worth to explain it, and discuss its choice in the first place. First of all, the standard more or less prescribes the use of Petri-nets as the semantic domain by stating that “use a Petri-like semantics” (cf. [21, p. 292]) (see e.g. [20, 26] for introductions). However, many of the constructs introduced in the standard are not present in elementary P/T-nets. Fortunately, most of these questions have already been addressed in the world of Petri-nets, e.g., high-level inscriptions occur in colored petri nets (cf. [17]), procedure calling has been treated in procedural nets (cf. [18]). All that is missing is a slight extension of procedural nets to cover exception handling and a fusion of the isolated features. This has been done in [31, 32], and due to lack of space, it is not possible to repeat the complete formal definition here.

A procedural net is a set of simple nets with initial and terminal markings, and a refinement function from transitions into the set of nets. Whenever a refined transition $t$...
fires, a \( \ell_{\text{call}} \) event is recorded and a new instance of the refinement net are created. When this instance reaches its terminal marking, a \( \ell_{\text{return}} \) event is recorded, and the instance is removed.

## 4 ConditionalNodes

The standard gives no hint on the concrete syntax of a ConditionalNode. I propose to represent it similar to an ExpansionRegion as a dashed line with ObjectNodes for the input- and output-parameters and one compartment for each pair of condition and consequence, separated by dashed lines (see Figure 4, left).

A conditional nodes is a kind of set of guarded commands: “a conditional node is a structured activity node that represents an exclusive choice among some number of alternatives.” (cf. [21, p. 313]). Each consists of a set of “clauses, [each consisting] of a test section and a body section”. When executing a ConditionalNode, all tests are executed. Then, the body section of one of those that yielded true is chosen nondeterministically and executed. Alternatively, “sequencing constraints may be specified among clauses”.

In order to reduce complexity, I propose that the test sections be side-effect free. Thus, it is probably the easiest to use the guards introduced with DecisionNodes. Also, I suggest that the body section consist of single Actions. These may call upon other Activities, of course. Then, ConditionalNodes are just syntactic sugar, and their meaning is best defined by an expansion into more basic constructs. Figure 4 (right) shows how this may be done.

\[ \text{Fig. 4. ConditionalNodes are syntactic sugar: sugared form (left) meaning (right).} \]

## 5 LoopNodes

A LoopNode has “setup, test, and body sections” (cf. [21, p. 341]). The standard gives no hint on the concrete syntax of a LoopNode. I propose to represent it similar to an ExpansionRegion (and my proposal for ConditionalNodes) as a dashed line with ObjectNodes for the input- and output-parameters and compartments for setup, test, and body regions separated by dashed lines (see Figure 5 (a)).
There are several possible interpretations of LoopNodes. The first interpretation considers the setup and body compartments as individual Actions (that might be refined by Activities). The test compartment becomes a guard on a DecisionNode, similar to the treatment of guards in ConditionalNodes (cf. Figure 5 (b)). A second interpretation might translate each of the compartments into a single action (cf. Figure 5 (c)).

![Diagram of LoopNodes interpretations](image)

**Fig. 5.** LoopNodes as syntactic sugar: sugared form (a) and possible expansions (b–d).

The standard provides a facility to implement both while-do and repeat-until loops, that is, loops where the condition to continue execution is tested before or after executing the loop body (cf. (c) and (d) of Figure 5). If the former case is intended, the isTestedFirst-attribute of LoopNode is set to true, in the latter case, it is set to false. Again, there is no syntactic feature in the standard to distinguish these two variants. Thus, I propose to use the sequence of the body and test sections in a LoopNode, that is, if the test section stands above the body section as in Figure 5 (a), the test is executed first, and we have a while-loop. For until-loops, the sections are simply interchanged. This would avoid an additional inscription, and make the type of loop very obvious even for people that are not familiar with the underlying concepts.

There are more questions concerning the interpretation of LoopNodes, however. The standard declares that a LoopNode “is a costructured activity node that represents a loop with setup, test, and body sections.” (cf. [21, p. 341]). Unfortunately, the meaning of the word “costructured” remains opaque. The standard goes on by saying that “each section is a well-nested subregion of the activity whose nodes follow any predecessors of the loop and precede any successors of the loop.” (cf. [21, p. 341]). This seems to indicate that a LoopNode is just a kind of additional structure on top of an Activity. That is, the Activity diagram fragment shown in Figure 6 (left) is identical to the Activity diagram fragment of Figure 6 (right), and the LoopNode structure shown there has no meaning at all, and is just a kind of reading guide that overlays the structure of the Activity. But then, why do we have LoopNodes in the first place?

Also, consider the interaction of loops with exceptions. A natural construction one would expect to be admissible (and simple enough semantically) is the situation shown in Figure 7 (left). What is the scope of the exception—is the body section also an Interrupt-
Fig. 6. LoopNodes in context: a fragment of an Activity containing a LoopNode (left), interpreting a LoopNode as an Action calling an Activity as described in Figure 5 (middle), and an alternative interpretation where the LoopNode is expanded in its context (right).

ibleActivityRegion? Or should it be possible that the exception is raised in the test and/or setup, too?

Actually, both situations make sense. So I propose to follow the interpretation shown in Figure 6 (middle) where the LoopNode is a plain Action refined by the net of, say, Figure 5 (d). This node is now also the protected node. So, it is possible to raise an exception anywhere in the loop. If the scope is supposed to be more restricted, the Activity that refines the “looping action” node raises the exception.

Fig. 7. Interactions between structured nodes and exceptions (a); the context of an ExpansionRegion (b) may call upon an Activity defined, say, like (c). In this example, the raising scope of the exception (the InterruptibleActivityRegion) has been set to encompass both the body and the test section.

Another interesting consideration is which result executing a LoopNode yields. The standard declares that “the test and body sections are executed repeatedly until the test
produces a false value. The results of the final execution of the test or body are available after completion of execution of the loop.” (cf. [21, p. 341]). This does not make much sense, however, since the last thing executed before leaving the loop must always be the test, and the test must finish with value false, so the result of a LoopNode is the constant false. It is not clear, how a result of the last execution of the body or an accumulated result may be passed on outside the LoopNode. This issue definitely needs clarification in the standard.

6 ExpansionRegions

The description of ExpansionRegions in the UML standard exhibits far more errors and inconsistencies, and is much less detailed than the descriptions of LoopNodes and ConditionalNodes. Also, there is a fairly straightforward intuition in terms of their practical value for the latter two, even if this is not documented in the standard. For ExpansionRegions, however, such a pragmatic intuition is not obvious. So, to some degree, this section is speculative in exploring possible intentions of the standard.

ExpansionRegions are ActivityNodes that process collections of elements as a unit. The standard declares that when “an execution of an activity makes a token available to the input of an ExpansionRegion, [it] consumes the token and begins execution. The ExpansionRegion is executed once for each element in the collection” (cf. [21, p. 326]). Thus, an ExpansionRegion can be viewed as a kind of map-function from one collection to another. Such a functionality may be implemented in four different ways (not counting mixtures of these).

iterative that is, in a loop where each of the elements is treated in turn, but processing starts only on the complete set of arguments;
streaming is similar to iterative in that only one element is processed at once, but processing of the first element may start even though further elements have not yet arrived;
parallel meaning that all elements are processed in lockstep, starting, proceeding, and ending together, irrespective of the actual time needed to process individual elements, and so possibly creating idle times;
concurrent is similar to parallel in that all arguments are treated potentially at the same time, but independent of each other rather than in parallel.

Following the UML standard, it is possible to specify three of these variants by setting the mode-attribute of ExpansionRegion. It may carry the values iterative, stream, and concurrent. Beware of the last mode setting, however: due to a spate of serious printing mistakes in the standard, the word parallel is used in all but one place instead of the word concurrent. The behavior explained is definitely concurrency, not parallelism, though: “the execution[s of the elements in the collection] may happen in parallel, or overlapping in time, but they are not required to” (cf. [21, p. 326]). To achieve parallelism, the standard would have to specify the execution as “parallel in lockstep, beginning, proceeding, and ending simultaneously” or similar.

At this point, we hit on an important feature of Activities. The standard states that “the expansion region is executed once for each element in the collection (or once per element position, if there are multiple collections).” (cf. [21, p. 326, emphasis added]) This seems to imply that an Activity is a kind of dataflow-computer with different parameters flowing through it—similar to a Petri-net, in fact³, and not like an individual run. Observe also,

³ Also, this raises again the question how well Activity Diagrams are suited for, say, workflow modeling, and whether Activities represent a workflow type/schema or an instance.
that if there are several computations going on at the same time, these must be isolated from each other to avoid interactions. Thus, a macro-like expansion strategy, as has often been proposed for high-level Petri-nets, is out of the question here. In [29], this problem has already been solved silently by the very definition of procedural Petri-nets. So, using the procedure call semantics for LoopNodes and ExpansionRegions, this problem is avoided altogether.

Thus, similar to the treatment of LoopNodes as proposed in Figure 6 (middle), ExpansionRegions should be translated as refined Actions (see Figure 7 b and c), where various calls to the same refinement transitions executes in its own state space. Depending on the mode of the ExpansionRegion, different refinement nets must be used (see Figure 9).

6.1 Iterative ExpansionRegions

In an iterative ExpansionRegion, “the executions of the region must happen in sequence, with one finishing before another can begin. […] Subsequent iterations start when the previous iteration is completed. During each of these cases, one element of the collection is made available to the execution of the region as a token during each execution of the region.” (cf. [21, p. 326]). This mode is treated semantically as shown in Figure 9 (left).

There, for simplicity, the collections have been implemented as lists. In the net, the region is represented by a transition. It may be refined to call another net in the sense of procedural Petri-nets (cf. [18, 29]). The region may start processing the first element of the collection right away by taking it from the list and, after processing it, adding it to a result list. When all elements have been processed, the Input collection is depleted (i.e. has become the empty list), and the resulting list is reversed to achieve the original order again (this last step may be omitted of course, if the collection is unordered).

6.2 Concurrent ExpansionRegions

In a concurrent ExpansionRegion, “the execution may happen in parallel, or overlapping in time, but they are not required to.” (cf. [21, p. 326]). This mode is treated semantically as shown in Figure 9 (right). There, the collection is first split up into its elements. The number of elements is tracked in the place Counter. Then, each element may be processed by the Region. Observe, that a transition may fire concurrently to itself as often, as there are tokens activating it. As soon as the first results are produced, they may be collected again by the join collection transition. If all results are processed, the Counter has been decreased to zero, and the output may be produced.

Note that it would be not easy at all to realize a truly lockstep-parallel execution mode of an ExpansionRegion, since we could not use the procedure call mechanism, but would have to resort to a kind of net folding.
At this point, a small digression concerning the tool used is in place. The nets in Figure 9 have been drawn and simulated using CPN Toolset (see [7]). The inscriptions are Standard ML code (the programming language used for inscriptions in the CPN Toolset, cf. [22]), with some special conventions. For instance, E is the color of plain tokens (“black dot tokens”), and the only value of this type is e. Multisets are represented like 1’true++2’false, which means: one token of value true and two tokens of value false. Tuples are written as (x, y), and lists as (head::tail).

6.3 Streaming ExpansionRegions

In an ExpansionRegion with mode stream, “there is a single execution of the region, but its input place receives a stream of elements from the collection. The values in the input collection are extracted and placed into the execution of the expansion region as a stream [...]. Such a region must handle streams properly or it is ill defined. When the execution of the entire stream is complete, any output streams are assembled into collections of the same kinds as the inputs.” (cf. [21, p. 326]). Thus, at any given time during the stream processing, some elements of a stream may have been processed already, some may be being processed in the very instant, and some may still be awaiting processing. This is in contrast to other collection-valued ExpansionRegions, where all elements of a collection must be present before the processing starts, and all elements of the collection must be processed before the Region is terminated.
Starting with the simple case of an individual stream, this behavior may be captured
by the net shown in Figure 9 (middle). Collections may be represented by tagging the
elements with sequence numbers, i.e. 

\[ \text{Element STREAM} = \text{Element} \times \text{SequenceNumber} \]

First of all, the stream is split into sequence numbers and elements proper. While the
element is processed by the Region, the sequence number is passed by. The complement
place ensures proper synchronisation.

As a side remark, the standard also proposes stream-valued Pins that are not
collection-valued (cf. [21, Fig. 283, p. 358], reproduced in Figure 8 a and b), but it re-
mains unclear, how this may be interpreted.

7 Conclusion

7.1 Summary and contribution

In this paper, the concepts related to StructuredActivityNodes are examined by defining
a straightforward semantics based on Petri nets. Some problems concerning the concrete
and abstract syntax and the semantics have been uncovered in the standard (concrete syn-
tax of Conditional- and LoopNodes, scope of exceptions, single/multiple inputs), and pos-
able solutions have been proposed.

There have been several proposals for semantics of Activity Diagrams, but most of
these aim at UML 1.x. For UML 2.0, there are only [3] and [29–32]. The current paper
covers structured nodes. Together with [29–32], now all of UML 2.0 Activities have been
covered with formal semantics (consider again Figure 1).

7.2 Related work

Since the UML standard has been written from scratch as far as Activity Diagrams are
concerned, most of the previous work examining UML Activity Diagrams (see [1, 2, 4–6,
8–15, 19, 23, 24, 27]) has become obsolete. See Figure 10 for a comparison.

In particular, structured nodes which have not been there in the UML 1.5 have not
been addressed so far. Also, it seems that so far, only very little has been published on the
UML 2.0 Activity Diagrams: [3] examines expansions and streaming in an intuitive way,
focusing on shared input pins and some aspects of streaming. [29, 30, 32, 31] provide for-
mal definitions of the semantics of control-flow, procedure call, data-flow, and exceptions
in UML 2.0 Activities, respectively. This paper builds on the latter four.

7.3 Open questions

We may thus now turn to questions like what refinement and composition means for Ac-
tivities, or how the new constructs work in the field. Also, the combination with other parts
of the UML must be examined, in particular the relationship to Interactions and StateMa-
hines, whose natural semantic domains must be related to Petri-nets. Also, examples
soon become too complex for manual treatment, and so we need tool-support.

Furthermore, if an ExpansionRegion is indeed a kind of map, then how would a fold-
operation be accomplished? It requires several flows to be joined together, but neither
are there specific constructs for this, nor are there any hints how several instances of one
Activity might interact. Note that it is not sufficient to simply inject a fork right at the
start of an ExpansionRegion. Embedding parameters in control-objects might be a way
out, but seems to raise more questions than it answers.
### Comparative Categorization of Previous Work Leading to This Article

The table below categorizes the previous work leading to this article (in column "control-flow", wf means well-formed, and nwf means non well formed. The degree of rigour is approximate, where a completely formal definition is high rigour, examples and some formalism is medium, and mere text is low.

<table>
<thead>
<tr>
<th>Authors, references</th>
<th>UML version</th>
<th>Semantic domain</th>
<th>Control flow</th>
<th>Data flow</th>
<th>Hierarchy</th>
<th>Exceptions</th>
<th>Streaming</th>
<th>Structured nodes</th>
<th>Rigor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allweyer et al. [1]</td>
<td>0.9</td>
<td></td>
<td>wf</td>
<td>√</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>low</td>
</tr>
<tr>
<td>Aprille et al. [2]</td>
<td>1.x</td>
<td>LOTOS</td>
<td>wf</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>medium</td>
</tr>
<tr>
<td>Börger et al. [6]</td>
<td>1.x</td>
<td>ASM</td>
<td>wf</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>-</td>
<td>medium</td>
</tr>
<tr>
<td>Bolton &amp; Davies [5,4]</td>
<td>1.3</td>
<td>CSP</td>
<td>wf</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>low</td>
</tr>
<tr>
<td>Eshuis &amp; Wieringa [9, 10]</td>
<td>1.x</td>
<td>algorithm</td>
<td>wf, nwf</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>high</td>
</tr>
<tr>
<td>Eshuis &amp; Wieringa [12, 11]</td>
<td>1.x</td>
<td>LTS</td>
<td>wf, nwf</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>high</td>
</tr>
<tr>
<td>Gehrke et al. [15]</td>
<td>1.0</td>
<td>PN</td>
<td>wf, nwf</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>medium</td>
</tr>
<tr>
<td>Pinheiro da Silva [24]</td>
<td>1.x</td>
<td>Lotos</td>
<td>wf, time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>low</td>
</tr>
<tr>
<td>Rodrigues [27]</td>
<td>1.x</td>
<td>FSP</td>
<td>wf</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>low</td>
</tr>
<tr>
<td>Li et al. [19]</td>
<td>1.x</td>
<td>LTS</td>
<td>wf</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>high</td>
</tr>
<tr>
<td>Störle [29]</td>
<td>2.0</td>
<td>PPN</td>
<td>wf, nwf</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>high</td>
</tr>
<tr>
<td>Störle [30]</td>
<td>2.0</td>
<td>CPN</td>
<td>wf, nwf</td>
<td>√</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>high</td>
</tr>
<tr>
<td>Störle [31, 32]</td>
<td>2.0</td>
<td>ECPN</td>
<td>(wf, nwf)</td>
<td>(√)</td>
<td>(√)</td>
<td>√</td>
<td></td>
<td>-</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Abbreviations:**
- PN = simple P/T-Petri-nets
- ECPN = exception colored Petri-nets
- FSP = finite state processes
- PPN = procedural Petri-nets
- LOTOS = Language of Temporal Ordering Specifications
- LTS = labeled transition systems
- CSP = communicating sequential processes
- ASM = abstract state machines

**Fig. 10.** Comparative categorization of the previous work leading to this article (in column "control-flow", wf means well-formed, and nwf means non well formed. The degree of rigour is approximate, where a completely formal definition is high rigour, examples and some formalism is medium, and mere text is low.)
References


UML-Based Testing: A Case Study

Marko Jäntti, Tanja Toroi

Department of Computer Science
University of Kuopio
P.O.B 1627
FIN-70211 Kuopio, Finland

Abstract. UML-based testing is a promising testing approach that enables software tester to collect information about the structure and the functionality of the system under test efficiently. UML diagrams from the project's specification phase form a natural basis for the test model. In this paper, we present results of a case study where we applied UML test model and component-based testing in order to test a healthcare application. This case study emphasizes user-side testing. Our findings from experiments were that using test model together with equivalence classes revealed several serious defects from the system after the developer testing. However, establishing a UML test model for the legacy system with insufficient documentation is a big challenge and requires that software testers have a good experience of UML modeling and enough domain knowledge.

1 Introduction

The UML-based testing has become a very popular testing method among test researchers and software companies. Various UML diagrams can be exploited in the system development. Activity diagrams are used to model human workflow and communication between the system and human actors. Use case diagrams show how users interact with the system. They provide a view of what the system does and identify the actors and the users of the system.

In the UML-based test model test cases are designed based on UML diagrams and supporting specification documents. According to Rational Unified Process [7] a test model is a description about what should be tested and how testing should be executed. RUP can be seen as a framework of software engineering process that uses UML notation. The test model of RUP includes test cases, test procedures (instructions for preparing, executing and evaluating testing), test scripts that automate test procedures, test classes and components (drivers etc.), test collaborations (connections between test components) and notes (constraints of the test model and other useful information for the test model).

Why do we need a test model? Software testing must be systematic, focused and automated [2]. A test model fills all of those three requirements by supporting classification of test input and state combinations of the system, focusing on input combinations that are likely to have defects and defining the manual testing process that is a prerequisite for automated testing. The main idea of UML test model is to derive test cases from system’s functional specifications [9, 3]. Many researches have pointed out the advantages of the model-based testing [9, 10, 11]. UML state diagrams have been used for automated generation of test cases [4] and sequence diagrams for static checking and test generation [1]. Yoon et al. have proposed UML-based test model for component integration testing using use case diagrams, sequence diagrams and collaboration diagrams to describe the functionality of the system [10].
In this paper we will describe our case study where we applied a UML test model and component testing to test a healthcare application. A black-box testing method was used in component testing because the program code was not available. Our testing model is focused on behavioral diagrams such as activity diagrams, state diagrams and use case diagrams because the primary testing objective in this study is functional testing (user-side testing) as an opposite to the developer-side testing.

The purpose of the case study is to evaluate UML-based test model in practice. Evaluation is based on using a real-world system as a testing target. Developers of the system were not doing UML-based testing activities. Evaluation criteria were set before the study. Criteria consisted of following questions: Can UML-based test model be used for testing an existing system? What are the parts of the UML model that are suitable for testing? What kind of defects can be found with a UML test model? A case study is used as a research method because the main objective of this study is to test the UML-based testing theory in practise, not to develop working practises of health care organizations (action research) although both objectives are interesting.

We will first present our results and then evaluate the model. The remainder of the paper is organized as follows: In Section 2 we describe research methods and tools used in this study. A case study of UML-based test case generation is described in Section 3. In Section 4 we examine test cases of different granularities of components. In Section 5 we evaluate the UML test model based on results of our case study. Finally, a conclusion is given in Section 6.

2 Methods and Tools

The purpose of the empirical study was to evaluate the UML-based test model in practice. We collected data for this research in a case study related to PlugIT project (a research project for application integration in healthcare, in Finland). A large national healthcare organization provided us usernames and passwords to their test environment. The test environment included a healthcare system that was responsible for maintaining patient information and referrals, reserving resources, etc.

At first, the functionality of the system was studied and analyzed through the user manual that had been sent us before hand. Activity diagrams and use case diagrams were drawn to describe the functionality of the system. After that, test cases were created based on the user manual and UML diagrams (for example Insert a patient). Parameter values for each test case (Insert a patient with a name “Tim Tester”) were generated by using equivalence classes: test inputs were classified into three groups (valid, invalid and illegal inputs). In a testing phase the test cases were executed manually in the test environment. The summary of the evaluation for using UML-based test model is presented in Section 5.

UML diagrams were created by Rational Rose Modeling Tool. Rational TestManager was planned to act as a test datastore for test cases but a text processing tool proved to be more useful because the health care organization did not have Rational's product licenses. UML diagrams are categorized in structural and behavioral diagrams. Our testing model focused on behavioral diagrams such as activity diagrams, state diagrams and use case diagrams because the primary testing objective in this study was functional testing. Static diagrams and sequence diagrams were excluded because they were not suitable for end users and there was no documentation of the healthcare system structure or object messages available.

We decided to focus modeling on two functions of the application. Both functions (Search a patient and Enter a Referral) are often used functions in healthcare applications. UML models were created by authors and PlugIT working group and presented to the healthcare organization in project meetings for validation.
3 UML-based Test Cases: a Case Study

Activity diagrams are useful for modeling human workflow and communication with the system and each other. Compared with use case diagrams, an activity diagram can show the order of use cases: some activities must be done before other or they can be done in parallel. An activity diagram for Search function is shown in Figure 1.

![Activity Diagram for Search a Patient function](image)

**Fig. 1.** An activity diagram for Search a Patient function

The diagram in Figure 1 showed that Search function included three major flows that needed to be tested. If the search result was 0, the system displayed a message Patient not found. If the result was more than 1, the system showed a list of patients to be selected and if the result was 1 the information about the patient was directly on the screen. For testing purposes the abstraction level of the activity diagram was unfortunately too high. In order to design effective test cases for Search function one should collect detailed information about search criteria that is not visible in the activity diagram. In our target application it was possible to search patients by eight different properties and combinations of them (last name, first name, date of birth etc.).

A state diagram is needed if the system includes entities (classes, objects or components) which have important states in the domain. We found that a state diagram suited very well to model various states of referrals because the states were already visible in a user manual. The state transition table was established based on the state diagram. Then, test cases were derived based on the state transition table. The state diagram in Figure 2 describes various states of incoming referral in a healthcare system. The first state is signed. Then, in a normal case the referral is just saved (state = approved) in the system. Sometimes referrals are sent to the wrong person and must be forwarded to the third party. In this point, we noticed that the test target consists of both states and substates (for example, the state Closed had substates Forwarded and Returned). The Returned state is needed when the user returns an internal referral within the organization. There are situations where the receiver of the referral needs more information about the subject of care than the referral includes: if one responds to the
referral and the type of response has been set as Request for supplement, the substate will change to a Request for supplement.

![State Diagram for Referral](image)

**Fig. 2.** A state diagram for a referral

After the most important states and transitions within the system had been modeled a state transition table (Table 1) was created. The start state is the state before any action, the actions are inputs from the user and the end state is the state after actions.

**Table 1.** A state transition table for a state diagram of Fig. 2

<table>
<thead>
<tr>
<th>Id</th>
<th>Start state (Referral)</th>
<th>Action</th>
<th>End State (Referral)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>There is an incoming referral.</td>
<td>Choose Save in Menu</td>
<td>State = Signed</td>
</tr>
<tr>
<td>2</td>
<td>State = Signed</td>
<td>Choose Save in Menu</td>
<td>State = Approved</td>
</tr>
<tr>
<td>3</td>
<td>State = Signed</td>
<td>Choose Forward in Menu</td>
<td>State = Closed Substate = Forwarded</td>
</tr>
<tr>
<td>4</td>
<td>State = Signed The referral is an internal referral.</td>
<td>Choose Return in Menu</td>
<td>State = Closed Substate = Returned</td>
</tr>
<tr>
<td>5</td>
<td>State = Signed</td>
<td>Reply to the referral, Type of response = Req. for supplement</td>
<td>State = Closed Substate = Req. for supplement</td>
</tr>
</tbody>
</table>

Test cases can be derived from a state transition table as follows: the start state is a prerequisite for a test case, actions are test inputs, end states are expected results. The test case table also includes actual results of testing and information about whether the test was failed or passed (F/P).

**Table 2.** A test case table derived from the state transition table

<table>
<thead>
<tr>
<th>Prerequisites</th>
<th>Input</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result</td>
<td>State is Signed</td>
<td>State is Approved, P</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------</td>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>1</td>
<td>There is an incoming referral.</td>
<td>Choose Save</td>
<td>State is Signed</td>
</tr>
<tr>
<td>2</td>
<td>State is Signed</td>
<td>Choose Save</td>
<td>State is Approved</td>
</tr>
<tr>
<td>3</td>
<td>State is Signed</td>
<td>Choose Forward</td>
<td>State is Closed Substate is Forwarded</td>
</tr>
<tr>
<td>4</td>
<td>State is Signed, The referral is internal.</td>
<td>Choose Return</td>
<td>State is Closed Substate is Returned</td>
</tr>
<tr>
<td>5</td>
<td>State = Signed</td>
<td>Reply to the referral</td>
<td>State is Closed Substate is Req. for supplement</td>
</tr>
</tbody>
</table>

A state diagram was very useful as a test model because it visualized the options of users' actions better than a long textual description in the user manual. The tester (even without any experience of UML) is able to see the state transitions that should be tested and test inputs related to transitions. In our case study, state-based testing revealed one serious defect (Run-time error 6160) in a Resource Management module and two defects (Run-Time error 438) in a Referral module. The run-time error 6160 was found by entering a negative value in Estimated time of care field when a Department care button was pressed down and an Outpatient care button was up. The run-time error 438 was found by selecting two same values for the attachment field (Attachment 1: X-Ray, Attachment 2: X-Ray). These defects could have been found earlier if the developer had tested the system more comprehensively.

### 4 Testing Components

#### 4.1 Test Case Definition

Herzum and Sims [5] have defined components of different granularities in layered architecture. We utilize their ideas in this paper. The granularity hierarchy means that a component-based system (CBS) is a composition of business components (BC), which in turn are compositions of the lowest level components.

In this section we will first define test cases based on different granularities of components. Then we will give some simplified examples according to the definition. A test case is generally defined as input and output for the system under test. A test case is defined in Rational Unified Process [7] as a set of test inputs, execution conditions, and expected results developed for particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. The granularity aspect must be taken into considerations when defining test cases. Components are of different granularities so different kinds of test cases are needed. Test cases form often a flow of test input output pairs, which follow each other in defined order. We extend the definition by granularity and flow aspects. Flow aspect is especially important: At the CBS level workflows of stakeholders should be tested in order to achieve goals of co-operating organizations and actors [8].
The test case is a set of test input and expected result pairs, and execution conditions:

- An ordered sequence of test input and expected result pairs at the component-based system level form an action flow between business components and co-operating users.
- An ordered sequence of test input and expected result pairs at the business component level form an operation flow between the lowest level components and one user.
- An ordered sequence of test input and expected result pairs at the lowest component level form a method flow between object classes inside the component.

4.2 Test Cases Based on Action Flow

Coarse-grained test cases at the component-based system level are derived from the activity and use case diagrams. Use case diagrams usually show only the communication between users and a software system. Thus the co-operation of human actors is not presented. We propose that the definition of use case diagram and activity diagram are extended to contain human interactions as well as automated actions (Korpela et al., 2001). Consequently, it is possible to test that human workflow fits together with actions of CBS. Building action flow based test cases consists of the following phases: Firstly, activity diagrams and use case diagrams are created. Secondly, action flows from these diagrams are derived. Thirdly, test cases for the designed action flows are created.

Use case diagrams show how users interact with the system. They provide a view of what the system does and identify the users of the system. In this paper, we derive action flows based on information provided by activity diagrams, use case diagrams and use case scenarios. In figure 3 there is a use case diagram for the Lab Test CBS. Human actors communicate with the component-based system that is a collection of business components (Patient BC, Lab Test BC, etc). Each component has its own responsibilities e.g. Patient BC is responsible for creating new patients, searching existing patients, and showing patient information. Human interaction, take a sample, is illustrated by a grey use case.

![Use case diagram for the Lab Test CBS](image)

Fig. 3. A use case diagram for the Lab Test CBS
After the use case diagram is generated action flows can be created. An action flow test input of Lab Test CBS could be the following (actors in parenthesis):
- Choose a patient; (user, Patient BC)
- Examine patient and save patient record; (user, Patient BC)
- Create a lab order; (user, Lab Test BC)
- Send the lab order to the lab and to the patient; (user, Lab Test BC, Patient BC, Department BC)
- Reception; (user, Lab Test BC)
- Take a sample; (patient, laboratory technician)
- Analyse the sample; (user, Test Result Analyser BC)
- Save and send lab test results and reference values; (user, Lab Test BC, Test result Analyser BC)
- Find lab test results and reference values; (user, Lab Test BC)

If lower levels have been tested carefully testing amount at CBS level decreases. We should, however, remember that the most important action flows of CBS should be tested. In table 3 we have a test case for testing the above workflow. The test case is in the highest granularity level.

Table 3. Test case for the action flow

**Input for the whole test case:** Patient that needs health care  
**Expected result for the whole test case:** Lab test results of the patient are available to the health professionals dealing with the patient  
**Precondition:** Lab tests are needed  
**Invariant:** There is a proper implementation for data security and safety  
**Post condition:** Lab test results of the patient exist in the system for the doctor  
**Environmental needs:** Lab test analyser is needed

<table>
<thead>
<tr>
<th>Step</th>
<th>Input</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Choose a patient</td>
<td>Search criteria and Search button</td>
</tr>
<tr>
<td>2.</td>
<td>Examine patient and save a patient record</td>
<td>Examination data about patient</td>
</tr>
<tr>
<td>3.</td>
<td>Create a lab order</td>
<td>Choose selected lab tests</td>
</tr>
<tr>
<td>4.</td>
<td>Send the lab order to the lab</td>
<td>Reserve time from lab. Send the lab order</td>
</tr>
<tr>
<td>5.</td>
<td>Reception</td>
<td>Choose a patient</td>
</tr>
<tr>
<td>6.</td>
<td>Take a sample</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Analyse the sample</td>
<td>Sample</td>
</tr>
<tr>
<td>8.1</td>
<td>Save lab test results and reference values</td>
<td>Lab test results and Save button</td>
</tr>
<tr>
<td>8.2</td>
<td>Send lab test results and reference values</td>
<td>Lab test results and Send button</td>
</tr>
<tr>
<td>9.</td>
<td>Search lab test results and reference values</td>
<td>Search criteria and Search button</td>
</tr>
</tbody>
</table>

4.3 Test Cases Based on Operation Flow
Test cases of medium-grained components consider operation flows of each actor using BC. In order to derive test cases at this level we need scenario descriptions besides use case diagrams. Building operation flow based test cases consists of the following phases: Firstly, use case scenarios are created. Secondly, operation flows from those scenarios are derived. Thirdly, test cases for the designed operation flows are created. The following use case scenario covers the Choose a patient use case (see Fig. 3).

**Use case:** Choose a patient

**Description:** A nurse chooses a patient

**Preconditions:** Patient record is needed

**Invariant:** A nurse has rights to look at patient information

**Post conditions:** Patient record is on the screen

**Basic flow:**
1. Enter search criteria (name, patient identifier).
2. Press Search button to execute a search.
3. Patient information is on the screen.

**Alternative flow:**
Several possible patient candidates found.
1. System returns a list of patients.
2. The nurse selects a patient from the list.
3. Patient information is on the screen.

**Exceptional flow:**
No candidates found.

=> Execute the use case: Create a patient.

Entered search criteria contain defects: an error message.

a) A user has not chosen any search criterion.
b) A user has entered special characters that are not allowed.

=> Make sure all the words are spelled correctly.

After the use case scenario is created operation flows can be defined. Next we give an example of an operation flow of Patient BC for the above use case:

− Input a patient number;
− Press Search button;
− Return a list of patients;
− Select a patient from the list;
− Return patient information;

When the operation flow is formed test cases based on it can be created. Test cases should be designed so that each path of the dependency graph of the BC will be traversed at least once. For example, for above operation flow test input should cover such patient numbers that returned list includes one patient, many patients, and none patients. A test case for the one flow of the operation flow (a list of many patients returned) is described in Table 4.

**Table 4.** Test case for the operation flow

<table>
<thead>
<tr>
<th>Step</th>
<th>Input</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enter a patient number.</td>
<td>The patient number is in a text field.</td>
</tr>
<tr>
<td>2</td>
<td>Press Search button.</td>
<td>A list of patients is returned.</td>
</tr>
<tr>
<td>3</td>
<td>Select a patient from a</td>
<td>Patient information is</td>
</tr>
</tbody>
</table>
Test cases should be designed and executed for the most important scenarios of the system, because large systems include a huge number of various scenarios and input combinations. The major problem is that all the scenarios seem to look important and it is difficult to identify critical and risky points from them. After the scenarios of high priority have been selected for testing each flow of the scenario is tested: a basic flow, an alternative flow and an exceptional flow. The exceptional flow plays an important role in preventing defects. It defines the most important ways to misuse the system. By defining exceptional situations the integrator tries to prevent users’ operations that could result an uncaught exception.

4.4 Test Cases Based on Method Flow

A component developer can derive the method flow from UML sequence diagram. Abdurazik and Offutt have formed message sequence path for a collaboration diagram using the messages and their sequence numbers [1]. In order to avoid restriction on object oriented systems we use the term *method flow* in this paper instead of a message sequence. The method flow can be derived from a sequence diagram. The following example of the method flow considers the user interface of the Patient BC (actors in parenthesis):

```
Sequence diagram: Search patient with a given selection criteria.
- Select Search patient method on user interface of Patient BC; (human)
- Invoke an interface implementation of GUI component; (component execution environment, GUI component)
- Handle invocation and pass it on Patient class; (Interface class)
- Invoke a proxy to target enterprise component; (Patient class)
- Send a message to the target, wait for the answer, and route it back; (proxy)
- If one patient is found, show information of this patient; (Patient class)
- If several patients are found, show a list of matching patients; (Patient class)
```

Above we have one method flow between classes inside the GUI component of Patient BC. Test inputs are selected so that at least one test input is selected from every equivalence class of patient search criteria. In this level dependency graph shows dependencies between classes. Thus we check that test cases traverse all the paths of all the dependency graphs formed. Although the case study did not include sequence diagrams with objects and methods we wanted to present the theory of granular test cases as a whole in this section.

5 Evaluation of UML test model

This paper has introduced results from a case study where we derived test cases from UML diagrams. We identified both advantages and problems in using UML test model. The advantages of UML-based testing found in this study are listed below:

**Advantages**

1. Use cases provide a way to divide the system under test into smaller functional units and test cases can be organized by use cases. Well-organized test case documents increase the quality of software product. A project customer is able to see from test documents how the system has been tested.
2. Use case scenarios include descriptions of exceptional and alternative flows that are often sources of defects.
3. Test cases of black-box level can be easily identified from the UML state diagram and the state transition table. Transition coverage can be used for measuring test coverage.
4. Activity diagrams show different action flows that a tester must go through in testing.
5. If someone has already created UML diagrams in a project, why not to take advantage of them as a test model. Visual modeling helps testers to understand the structure and behavior of the system in a shorter time than without models.
6. Software testing becomes more systematic with a test model. Our testing experiments for a healthcare application revealed three serious run-time errors and dozens of minor defects like warning dialogs displayed without any reason and inconsistencies in error checking of text fields.

Our study also revealed following problems related to UML-based testing:

**Problems**

1. Developers must understand the importance of UML models, deliver models to the customer and organize training. Often, there is no specification documentation of the system or related UML diagrams available. End users and software testers do not have time to draw UML diagrams in a testing phase and it is not their job to do that. Testers of the health care organization were able to test efficiently the normal flow of actions but testing of exceptional flows was insufficient.
2. UML diagrams are often too abstract for testing purposes. In many cases, the textual description of diagrams is a better source of test supporting information. For example, use case diagrams did not provide enough information to support software testing. Use case scenarios with detailed information were more useful.
3. Motivation. Testers may feel that establishing a UML test model for a legacy system is less exciting than establishing a test model for new software. Building and testing something new looks more attractive.
4. Prioritization. Because of limited testing resources, testing and test cases must be focused on areas where defects are most serious and areas that are more likely to have defects. Writing test lists, even for a simple application, requires time and patience.
5. Developers should execute testing activities more carefully and comprehensively to reduce the required testing work of the customer. Developers must perform white-box testing because the program code is available for them.

The diagrams that were excluded from this study can also support testing activities. The software developer uses class diagrams in order to describe the static structure of the system. The class diagram shows dependencies between classes. The method or message sequence between classes or modules is typically shown as sequence diagrams that describe the dynamic behavior of the classes. The software developer can use a component diagram to describe the dependencies between components. A deployment diagram models the physical architecture. Both hardware and software nodes can be included in a deployment diagram. After the software designer or developer has specified the system with UML diagrams from different viewpoints the software testers can use them as useful information for building a test model. The developer-side should test the system from all viewpoints of testing: the system architecture, the internal structure of components, interaction between components, functionality and performance etc. A customer or an end user typically tests only the functionality of system because the code is not available.

Use case diagrams, state diagrams and activity diagrams were selected for this case study because they were closer to the viewpoint of end user testing than sequence diagrams or class diagrams that belong to the viewpoint of developer testing. The starting point of the case study was to derive test cases based on a user manual of the system. It was harder than expected to create UML models afterward for a healthcare application. The models remained simplified examples. However, they supported testing activities and many new defects and improvements for the user manual were found.
6 Conclusion

Our results indicate that the UML-based test model has both advantages and problems. Test cases can be easily identified from the UML state diagram and the state transition table. Use case diagrams and scenarios include descriptions of exceptional and alternative flows that are often sources of defects. Activity diagrams show workflows that are important to be tested. End users were able to understand state diagrams, activity diagrams and use case diagrams without a previous experience with UML. One major problem seems to be that UML diagrams are often too abstract for testing purposes. A typical situation related especially to the legacy systems is that there is no specification documentation like UML diagrams available for software testers.

In some cases UML test model does not necessarily provide all required information for testing. If a tester wants to test a form in a graphical user interface, he/she must probably try input combinations (legal, illegal, invalid) that are not included in test cases derived from UML models. Therefore, it is recommendable to use other testing techniques like equivalence classes together with UML test model.

In this paper we also proposed that the granularity aspect guides testing component-based systems. Furthermore, we proposed that a flow aspect is important in test case design.

This study has contributed to the attitudes of software companies in PlugIT project on testing. They have realized that the results of the testing research can be utilized in software development. In further research we shall consider the business of actors in component-based software production more accurately. We will outline responsibilities, duties, working practices and content of contracts between co-operating actors, which are related to testing and error management. We will perform a case study based on a set of defects found in reusable components used in several applications and classify these defects in order to improve the component testing from the component developer viewpoint.

7 Acknowledgements

This paper is based on research in the PlugIT project (2001-2004), funded by the National Technology Agency TEKES grants no. 40664/01 and 40246/02. We wish to thank professor Anne Eerola for her comments and ideas.

References

6. Korpela, M., Eerola, A., Mursu, A., and Soriyan, HA.: Use cases as actions within activities: Bridging the gap between information systems development and software


Modeling Concern-Based Testing with UML Testing Profile

Mika Maunumaa and Mika Katar

Institute of Software Systems
Tampere University of Technology
PO Box 553, FIN-33101 Tampere, Finland
{mika.maunumaa, mika.katar}@tut.fi

Abstract Modern iterative and incremental software development processes require new types of modeling and testing support. Especially when developing systems one increment at a time, it is beneficial to group the associated test suite and test architecture together with the actual UML model of the increment. In this paper, we extend our previous work on aspect-oriented design with UML to support testing of such increments. We define our extensions in a manner that is compatible with the upcoming UML 2.0 Testing Profile. The approach is illustrated with a small example, which models testing of a feature interaction issue in an embedded system.

1 Introduction

In modern software engineering, it has become a standard practice to develop systems iteratively in small increments. The basic idea is to start by implementing a solid architecture on top of which features visible to users can be added one by one. Such a development process enables early interaction with the users and customers who can evaluate the results before the requirements have been implemented in full. Moreover, often too optimistic delivery schedules can be met by delivering a system that does not yet have all the advanced functionality but implements a basic set of features that is sufficient in most use cases.

Such iterative and incremental processes should be supported by testing and design techniques. For instance, it is important to trace back from each test case to requirements across different increments. Concerning conventional design notations, one problem is that the increments often cut across several classes, components, process etc. In other words, adding an increment, corresponding to a new feature, for instance, means adding new classes and possibly modifying several existing ones. Besides checking that the new feature integrates well with the previous features (integration testing), testing should be used to verify that the addition of the new feature has not broken the old implementation (regression testing).

On the one hand, iterative development means that each increment is potentially scattered around the class hierarchy. On the other hand, as the classes grow in each iteration cycle, the increments become tangled inside them. It has been speculated that scattering and tangling can cause problems with traceability, comprehensibility, maintainability, low reuse, high impacts of changes and reduced concurrency in development [5]. To alleviate such issues, several aspect-oriented design methods have been proposed (see [1]), which enable capturing cross-cutting concerns, i.e. conceptual matters of interest, such as those addressed in increments, in a modular way.
In traditional waterfall-based development, test execution can be started only after the system has been implemented in full. Even though different phases of testing, (e.g. system, integration and unit testing) can be planned carefully beforehand, test execution is the last phase of the project. To meet the stringent deadlines, testing is often cut down resulting in lower quality products.

In contrast, in an iterative development process (e.g. Rational Unified Process [14]), each new increment for a component should pass its test suite before the next iteration cycle can start for the component. This means that instead of being the last and the most undervalued phase of the project, testing becomes an essential and more integrated part of the development process. This trend is seen for instance in Test-Driven Development [2] style of programming, an offspring of the popular Extreme Programming, advocating an approach where test cases are coded before the actual code under test. In such test-first approaches, system documentation is commonly replaced by test cases utilizing unit testing framework such as JUnit [7].

In this paper, our aim is to integrate an aspect-oriented design notation with test modeling utilizing the UML 2.0 Testing Profile (UTP) [10]. On the one hand, aspects allow modeling increments in UML in traceable ways. On the other hand, the ability to associate each increment with the corresponding test architecture and test suite facilitates understanding the system just as in the test-first programming mentioned above. Furthermore, mappings from UTP into TTCN-3 and JUnit enable a straightforward way to implement the test designs.

The rest of this paper is structured as follows. Section 2 introduces the necessary background for our approach on concern-based testing in UML. In Section 3, the approach is illustrated by an example of an embedded system design involving feature interaction issues to be tested. Finally, Section 4 draws the conclusions and outlines the future work.

2 Motivation and Background

In this section, we will first outline our contribution on combining Concern Architectures [8] and UML Testing Profile [10] concepts to visualize Concern-Based Testing in UML. Then we will introduce the necessary background theory (i.e. Concern Architectures and UML Testing Profile) in more detail.

We adapt Sutton’s and Tarr’s [15] view to emphasize the distinction between concerns and UML artifacts implementing them. Concerns are conceptual matters of interest, such as user requirements, and can be treated by one or more aspects. On the other hand, the term aspect is used for a stereotyped UML package that is able to encapsulate design artifacts treating an otherwise cross-cutting concern. In the case of more than one aspect corresponding to a concern the aspects are referred to sub-aspects, which can be shared by more than one concern. However, sub-aspects can be composed to form a composite aspect matching a single concern.

### 2.1 Concern-Based Testing in UML

**Basic Idea**  Testware can be represented in UML much in the same way as software under test. Both are derived from the same source, i.e. requirements. Requirements are a composition of functional, non-functional, and usage needs. In simple terms, software design declares ‘it is so’ and testware design asks ‘is it so’.

In our paper, we define aspects as collections of closely related diagrams modeling some logical whole in the system. At the design level, we see testing as a concern that
encompasses the design of the system under test. The testing concern includes the aspects of the system under test and an aspect defining the associated test architecture and test suite. Specifying a concern-based test architecture is a fairly straightforward task. First, we need to have a concern-based model. The model may have been created conventionally and remodularized after that to concerns and aspects, or designed using them in the first place. Next, we need to select the concern that we are going to test and figure out what are our testing objectives for the concern. Finally, we need to make a concern composition, i.e., a flat model containing all the elements of the aspects inside the concern that is placed under test. Now we have a fragment of the design that has everything what is needed to address the concern in question, and in particular, enough information to define the corresponding tests.

**Processing Concerns to Test Model** The first step in defining tests is to make an initial test architecture. First, we need to obtain a class diagram that contains only boundary elements between the required and provided parts of an aspect. Boundary elements include all the elements of an aspect, except those provided elements that do not communicate with any required element. The boundary elements are candidates either for test stubs or drivers, or system interfaces, i.e. SUTs. From the composed model, we extract every class that has stereotype ‘required’ and every ‘provided’ class that has a connection to any of those ‘required’ classes. This model forms the test base model. Second, we need to extract all behavior that is related to those classes. This behavior gives us a reasonable basis for designing test behavior because it contains all communication that goes through the boundary elements with respect to this particular concern. Finally, we obtain a full test base model with structure and behavior.

To obtain a final test model, the test base model must be converted from the design world to the testing world. ‘Required’ classes that lack actual implementation are good candidates for test components. On the other hand, ‘provided’ classes that are connected to ‘required’ classes are good candidates for SUTs. Test cases are derived from behavioral diagrams and the actual purposes of the concern. Obtaining this information is not in the scope of this paper, but similar techniques can be found in [4].

The process described above can create an insufficient test model because it somewhat assumes that testability has been designed into the system. Because this is not usually the case, the final test model might need to be complemented with additional classes and behavioral elements from the test base model. Because it is difficult to come up with the best possible model at once, creation of a test model often becomes an iterative process.

### 2.2 Concern Architectures

Concern Architectures is a software architecture viewtype [6] for providing aspect-oriented views on software architecture. The basic idea is to view the system design as a composition of aspects that contain cross-cutting fragments of the total system. In more detail, each aspect consists of two parts, the required and the provided part. In the required part, the context in which the aspect is applicable is described. For instance, if an aspect adds methods to a class that some other aspect provides, the class must be required by the former aspect. The provided part, on the other hand, defines those elements that are introduced by this aspect. This way aspects can provide augmentations that cut across existing module boundaries.

Systems are built from aspects utilizing an asymmetric composition operation, which collapses and combines model fragments presented by aspects together. In the composition of aspects, one aspect augments another aspect. For instance, the aspect discussed
above defining new methods augments the one introducing the class for the methods. As an example, in Figure 1, a simple aspect with provided and required elements is shown. All the elements not marked as required are assumed to be provided. Among other things, the aspect requires two states of a state machine (states \textit{Play-on} and \textit{Alarm-on}) and provides a transition (firing event \textit{alarm}) between them.

A concern architecture comprises concerns, aspects, and dependencies in between them. The concerns are collections of aspects corresponding to some conceptual matters of interest for some stakeholder, such as features emerging from requirements. It should be noted that each aspect can belong to more than one concern. To view one concern as a whole, the aspects and subconcerns it contains are composed into a one composite aspect. The dependencies are formed when a partial order is imposed on the aspects, meaning the order in which they must be composed (because of the asymmetric composition operation, ordering is relevant). The dependency between an aspect and a concern is an abbreviation for the dependency between aspect outside and inside the concern.

The approach as such is generic and must be instantiated for some design notation before it can be applied. A profile instantiating the model for UML has been outlined in [9]. The profile consists of stereotypes, tag definitions, and OCL constraints\(^1\). In the profile, aspects and concern are stereotyped UML packages and tags are used to indicate required

\(^{1}\) The reason for using profiles instead of direct extensions to the UML metamodel is simply to assure better compatibility with other UML extensions.
and provided elements. OCL constraints are used to define well-formedness rules, ensuring for instance that concerns do not own model elements since they can be overlapping. Instead, all the contents of concerns are imported.

To illustrate the approach, in Figure 2, a concern architecture describing one part of a simple embedded system is shown. The system will be used as a running example in the sequel. The ’Play&Alarm’ aspect, shown in detail in Figure 1, depends on certain aspects some of which are included in the ’Playing’ and ’AlarmClock’ concerns as indicated by the dependency arrows. This means that ’Play&Alarm’ requires some elements from those aspects.

![Figure 2. AlarmWhilePlaying concern](image)

The idea of the concern architecture in this particular example is to highlight a feature interaction issue. The embedded device, adapted from [12], is a small dictating machine capable of recording short messages to a digital memory through a microphone and playing those messages through a speaker. Assuming that these basic features have been implemented in a low-end version of the product, we now want to introduce a high-end version that can also serve as an alarm clock.

However, an increment augmenting the system with the alarm clock functionality introduces feature interactions. In particular, it is not obvious what should happen if the alarm occurs while the user is playing or recording a message. In Figure 2, we have depicted the former situation with alarm while playing. The two concerns, ’Playing’ and ’AlarmClock’ have been included into a concern corresponding to the feature interaction called ’AlarmWhilePlaying’. The interaction problem is resolved in the ’Play&Alarm’ aspect (Figure 1) by giving precedence to alarm over playing using a sequence diagram and a state machine.
2.3 UML 2.0 Testing Profile (UTP)

The UML Testing Profile (UTP) [10] is an upcoming UML 2.0 compliant profile providing modeling concepts for black-box testing [3]. The profile defines means to develop testware architecture and behavior. In addition, it offers tools for test configuration management and extends original definitions of UML 2.0’s data and time concepts [11]. These concepts define a language for visualizing, specifying, analyzing, constructing and documenting the artifacts of a test system [10], and will be described in detail in the following.

Test Architecture The test architecture part comprises three fundamental concepts: test suite, test component, and system under test (SUT). Test suite defines a collection of related test cases and can be modeled as a stereotyped class. Each suite includes behavior (testcase), which controls the test case execution. Test component can models two concepts: test drivers and/or test stubs. A test driver is responsible for running the component under test as test sequence requires whereas a test stub is responsible for providing the response that is required to test some behavior inside the component.

Test components realize the test cases within the suite and execute them. They communicate with each other and, in particular, with SUTs. Test suite contains public test cases that are the interface towards the tester. It can also contain non-public test cases that are used as utilities when a public test case is realized, for instance, by dividing test cases into sub cases.

The SUT represents the system under test from the black-box testing point of view. It can be though as an adapter or a proxy between the system software and the testware. Each suite includes a test configuration (specified using UML deployment diagram [11]) to model the configuration of a particular test setting. It contains all the test components and SUTs that are relevant for the test suite.

There is also an interface called Arbiter that each test component should implement if it wants to have a veto right for the test execution. Each arbiter can set a verdict after each test run indicating whether the test case should pass or fail. For every test configuration, the master arbiter collects all the verdicts and sets the final verdict.

Verdict indicates the result of one test case execution. The value of a verdict can be one of the following: pass, inconclusive, fail, and error. Pass verdict tells that the SUT has functioned as expected and fail that it did not function as expected. Inconclusive verdict means that for some reason it is not possible to determine the result of the test. If the value of a verdict is error, then the test system itself has failed and is not able to continue test execution. Verdicts are exclusive, i.e. inconclusive surpasses pass etc.

Test Behavior The concept of test behavior corresponds to the dynamic side of testing. Shortly, it has three responsibilities: how to define a test sequence, how to justify the existence of a test case, and how to declare the judgment if a test run succeeded or not. Any UML 2.0 behavioral diagram type can define a test behavior, but in practice, interaction diagrams or state machines are the most useful ones for this purpose.

To document the intentions of the test designer there is a concept called test objective. The test objective is a stereotyped dependency showing the motive behind the test case, for example, a requirement or a use case.

One important concept defined by test behavior is test case. A test case is a specification of the interaction between test components and SUT in order to realize the test objective.

Another useful concept in the UTP is default behavior. Although the test case specifies the known action with the known response in interaction with a SUT, the possibility of
unspecified response from the SUT exits. To handle such unexpected situations the UTP introduces the concept of default action. The test designer can declare default action or can leave that to some higher-level default action handler.

To report how a test case execution is progressing, the test case can generate a test trace. A concept called log action can be used to generate logging information during the test execution.

As an example, Figure 3 illustrates a test architecture showing configuration for a particular test setting. Figure 4 illustrates the test behavior of the same test setting. In the case of any other sequence than the one described in the figure, the outcome of the test case is not 'pass'. The 'default' determines the verdict in these cases.

**Time** UML 2.0 does not contain time concepts adequate for testing purposes. The UTP declares such concepts including timer and time zone. Timers can control the execution of test cases. In addition, a timer can also terminate a test case if the SUT has not responded within given expiration time. In distributed systems, a time zone groups components together and provides them with the same perception of time. The comparison of time events from different time zones is illegal as only events within the same time zone are comparable.

Coming back to Figure 4, at first, the main test component initializes the alarm clock, starts the timeout timer 'T1', and requests 'Timer' component to start timing. Timeout is used as a guard for testing if 'AlarmClock' works improperly. If 'AlarmClock' alarms within expected time period, T1 will be stopped and test case obtains the verdict 'pass'.

### 3 Example

The selected features 'Play' and 'Alarm' introduce fairly trivial testing problems if tested individually. However, the combination of the two presents a different problem, which is not so trivial to test since it involves concurrent events.
The features follow from the following requirements: "DSR shall 'Play' the recorded message", "DSR shall sound 'Alarm' on designated time", and "DSR shall play 'Alarm' while playing message". The first two requirements are easy to point out from the model but the last one is more difficult.

In the following example, we apply our approach to create a test model for black-box integration testing. An important and fundamental feature interaction issue illustrated in Figures 1 and 2 gave us a reason to create a test system to verify the proper functionality associated with this concern.

3.1 Digital Sound Recorder Model

As already mentioned, the digital sound recorder (DSR) is a simple dictation machine that has basic features like recording and playing a message. In addition, the device has a capability to act as an alarm clock. The first two operations are sequential whereas the third operation has sequential part (setting and arming an alarm) and concurrent part (alarm activation).

System Model  The machine has a small LCD display to show information and options to the user as well as a few keys to navigate through those options and execute dictation commands ("Play", "Record", and "Stop"). In addition, there is a microphone for recording messages and a speaker for listening selected messages.

The DSR's system software has three distinct subsystems that separate different parts of the system: 'UserInterface', 'Audio', and 'Alarm Clock'. In addition to these, there are several supporting subsystems (see Figure 5).
The ‘UserInterface’ subsystem has one controller class ‘UserInterface’ that controls the interpretation of user actions to proper sequences of commands. These commands coordinate the corresponding subsystems to act accordingly. The ‘UserInterface’ class contains several view classes to display information as well as an association to class ‘Keyboard’ for user input. The response for each key press will be interpreted in various ‘UserMode’-based classes. A simple collaboration scenario for user interface is as follows:

1. 'User' presses the "Play" key,
2. 'Keyboard' detects user action and informs 'UserInterface',
3. 'UserInterface' informs current 'UserMode' object,
4. If the current 'UserMode' object is of the class 'MenuUserMode' and it is in state 'Message Menu', the 'UserMode' object will instruct the 'UserInterface' to start the 'Play' operation,
5. The 'UserInterface' object commands the 'AudioController' object to start 'Play Message' for currently selected message,
6. When 'AudioController' begins to play the message, it informs 'UserInterface' with 'AudioTask' event that the 'Play' operation has started, and
7. Whenever the 'UserInterface' receives the 'AudioTask' event, it shows the progress of the task in using object 'TaskView'.

Audio subsystem is responsible for recording and playing messages and alarm sounds. Audio subsystem’s the most important class is 'AudioController' because it contains all audio related application logic and all other classes within the subsystem supports 'AudioController' to do its task. The functionality of the subsystem is straightforward:

**Record**

1. 'Microphone' collects 'AudioSamples' and delivers them to 'AudioInput',
2. 'AudioInput' compresses samples and stores them into 'AudioBlocks',
3. 'AudioController' stores 'AudioBlocks' into a 'Message' object, and
4. The 'Message' object is stored into a 'MessageMemory' object.

**Play**
1. 'AudioController' takes the requested 'Message' object from 'MessageMemory',
2. Each 'AudioBlock' inside the 'Message' is sent to 'AudioOutput', and
3. Each 'AudioSample' inside the 'AudioBlock' is uncompressed and sent to 'Speaker'.

The Alarm clock subsystem contains the 'AlarmClock' class that maintains wall clock time and alarm time. The 'Timer' class keeps track of the passage of the time and updates 'AlarmClock' accordingly. Each second, it requests the 'AlarmClock' to set the current time and date. After time update, 'AlarmClock' instructs 'UserInterface' to show the updated value of time on the display. 'AlarmClock' also checks if there is a pending alarm. When alarm time occurs, the 'AlarmClock' informs 'UserInterface' about the occurrence of the alarm.

**Division into Concerns** In the DSR model, the system has been divided into subsystems in a conventional way. However, it is sometimes more useful to view parts of the subsystems as aspects that treat some concerns of interest, like the feature interaction in this case.

In concern architectures, an aspect collects those elements and diagrams together that form a logical whole. While aspects are a utility for creating design fragments, they need to be grouped into collections matching the concerns of interest.

![Figure 6. 'TestingAlarmWhilePlaying' concern architecture](image)

In order to illustrate how testing can be seen as a concern and how aspects can be used to encapsulate test suites for other aspects, we have extended the concern architecture seen in Figure 2. Firstly, we have created a new concern called 'TestingAlarmWhilePlaying' which includes an aspect called 'AlarmWhilePlayingTestsuite'. Both the concern and the aspect are illustrated in Figure 6. It should be noted that the new aspect depends on the 'AlarmWhilePlaying' concern. This is obvious, because the test architecture associated with the suite inside the aspect requires some classes and other elements from the concern under test.

**Modeling the Test** Before we can start the creation of the test model, we must have the system model corresponding our concern. The system model can be obtained by composing the aspects inside our target concern, 'AlarmWhilePlaying'. This composition results in a design fragment containing all classes that belong to the aspects. However, this model is too large for our purposes, and we have to extract only the necessary information.

To extract suitable model for testing, we must consider which classes provide interfaces to the system (SUT candidates) and which provide means to use those interfaces
Figure 7. Test model extracted from 'AlarmWhilePlaying' concern

(test component candidates). The detection of the latter from the composed model is quite simple. Every class having a stereotype 'required' must become a provided test component into the test model. System interface classes, i.e. the SUTs, can also be extracted

Figure 8. Final test model composed of the extracted model and the testing concern
from the composed model. As discussed earlier, a good candidate for a SUT is a 'pro-
vided' class that has some connection to the required classes.

The extracted model is shown in Figure 7. This class diagram is used to derive the final
test model. The final test model can be seen in Figure 8. It includes the main test compo-
nent and the test suite, as well as the associations needed. The model has been composed
of a test aspect, which provides 'TestPlayAndAlarm' test suite, and the 'Main' test com-
ponent, and it needs also other elements shown in Figure 7.

In contrast to class diagrams, which provide a basis for test structure, behavioral dia-
grams, such as sequence and state diagrams, provide a basis for test behavior in the test
execution phase. Both diagram types associated with a concern can be used for creating
test cases, corresponding to either legal or illegal behavior. From the behavioral diagrams
combined with a test objective, it is possible to determine a plausible set of test case
sequences.

![Test sequence for 'testPlayAndAlarm'](image)

**Figure 9. Test sequence for 'testPlayAndAlarm'**

A test sequence for testing the selected concern is illustrated in Figure 9. This is the
partial test sequence for test case 'testPlayAndAlarm'. Some messages between SUTs
and test components are removed to make the diagram more readable. The test case is
included in the test suite in aspect 'AlarmWhilePlayingTestsuite'. The test case itself is
divided into four utility test cases. When the whole sequence executes as illustrated in
the figure, the main test component sets verdict 'pass' for the test. In any other case, the
verdict can be set to 'inconclusive' or worse.
4 Conclusions

Test design is often based on a full implementation of the model, hence testing operability of more or less arbitrary cluster of classes. Testing should have some fundamental driver, like an important feature, that would justify focusing the effort. In our approach, this justification can be easily extracted from the concern-based model. The concern-based model combines only those model elements that are vital for the particular feature, thus making it easier to concentrate the testing effort to those model elements only.

We have presented an approach for modeling concern-based testing in UML using the UML 2.0 Testing Profile. The approach is based on our earlier work on so-called Concern Architecture viewtype providing aspect-oriented views on software architecture. The approach allows documenting increments of iterative software development in a way that illustrates both the increments and the associated test suite and architecture in an aspect-oriented fashion. We consider this approach to test modelling a promising technique, but it will need to be validated in the future in industrial case studies. In addition, tool support should be considered, because test modeling is labor intensive operation. Since both Concern Architecture and UTP are described as UML profiles, they can be processed with model processing facilities like xUMLi platform ([13]) or with scripts in modeling tools.

Presently our approach does not support extending the test models, which can be considered as a waste of resources. In the future, we are going to extend our approach to support reuse and extensibility of testware.

Acknowledgments

The authors would like to thank Tommi Mikkonen, Imed Hammouda and the anonymous reviewers for helpful comments on improving the manuscript.

References


HAZOP Analysis of UML-Based Software Architecture
Descriptions of Safety-Critical Systems

Klaus Marius Hansen¹, Lisa Wells¹, and Thomas Maier²

¹Department of Computer Science, University of Aarhus,
IT-Parken, Aabogade 34, DK-8200 Aarhus N, Denmark
{klaus.m.hansen, wells}@daimi.au.dk
²Danfoss Drives A/S, Ulsnaes 1
DK-6300 Graasten, Denmark
tm@danfoss.com

Abstract. Safety-critical systems which are systems that may harm the environment they are operating in are commonplace. Standards and regulations for developing software for safety-critical systems usually require restricted programming languages and formal methods. For reasons of scale and for coping with the complexities of large systems it may be beneficial to use modern object-oriented techniques in their development. A good match for such techniques is the semi-formal modeling notation Unified Modeling Language (UML). In order to use UML, the notation and associated techniques need to be reconciled with traditional techniques for safety-critical development. This paper looks at the commonly used Hazard and Operability (HAZOP) technique for hazard analysis and presents a systematic way of performing HAZOP on UML models together with a concrete interpretation of HAZOP for UML-based software architecture descriptions.

1 Introduction

Safety-critical systems are systems that can cause undesired loss or damage to life, property, or the environment, and safety-critical software is any software that can contribute to such loss or damage [13]. Object-oriented technologies are increasingly being used to develop safety-critical software. Some safety experts, international safety-certification authorities [8], and industrial guidelines [2,17] have expressed concerns about the lack of generally accepted and well-tested techniques, tools, and guidelines for developing safety-critical object-oriented software.

There is clearly a need for better techniques and guidelines for developing safety-critical object-oriented software. The process of developing safety-critical software encompasses a number of challenges that are not faced when developing non-safety-critical software. This paper focuses on one such challenge, namely software hazard analysis:

Software hazard analysis can identify ways in which a computer’s behavior and interfaces to the rest of the system can lead to system hazards. When critical behavior is identified, it can be traced to software design and code to identify parts of the software that require special design features or that need to be analyzed in depth. Because of the difficulty of [evaluating software safety], normally each part of the software development process is evaluated rather than just waiting until the software is complete [13, p. 302].

There are a number of techniques that are used for software hazard analysis, including Software Failure Modes and Effects Analysis (SFMEA) [20], Requirement Completeness
Analysis [14], and Hazard and Operability Studies (HAZOP Studies) [15]. HAZOP, which is summarized in Sect. 3.1, has particular promise as a technique for object-oriented software hazard analysis because it can be applied throughout the entire software development process to various different representations of the software and at various levels of abstraction. This coincides nicely with software development based on the Unified Modeling Language (UML) [19]. By performing HAZOP studies on UML models, safety analysis can be performed throughout software development, thereby increasing the chances for identifying critical behavior to be eliminated, alleviated, or analyzed in depth in later development stages.

The work presented in this paper was developed as part of a collaborative research project between the University of Aarhus, Danfoss Drives A/S 1, Systematic Software Engineering2, and ISIS Katrinebjerg3. The research goals of the project were to examine, compare, and develop new techniques for developing safety-critical, object-oriented software.

1.1 Related Work

This paper has been inspired, in part, by the work of [12], in which Lano et al successfully clarify some of the ambiguities regarding guidelines in [15] for applying HAZOP to two kinds of object-oriented models, and they provide additional guidelines for other kinds of object-oriented models. However, they do not provide a systematic set of guidelines for HAZOP of UML models as a number of UML diagrams and UML elements are ignored. Other authors have tried to formalize the UML in various ways in order to facilitate formal techniques of safety-critical software development [10]. While this is a rigorous alternative, software developers may be reluctant to use formalized UML just as they are reluctant to use formal methods in general. Moreover, it is unclear how well these approaches scale to larger systems. HAZOP studies have also been successfully performed on several kinds of software engineering models such a variant of Z [7], and data flow models and CORE models [16]. HAZOP is clearly useful for software hazard analysis, and safety-certification authorities recommend the use of HAZOP during software development [8], but few experience reports discuss how HAZOP can be applied to widely-used software engineering models. In addition to HAZOP, other safety analysis techniques have been combined with UML-like diagrams, such as Functional Failure Analysis combined with use cases [9], and Failure Modes, Effects and Criticality Analysis combined with Message Sequence Charts [6]. There are, however, no guidelines for applying these software hazard analysis techniques to all of the different kinds of UML diagrams.

1.2 Contributions

This paper provides a systematic alternative to the work presented in [12] while at the same time preserving the balance between formality and expressiveness inherent in UML models. Concrete examples will illustrate how HAZOP can systematically be applied to a selected subset of UML elements. The focus will be on the UML diagrams that are appropriate for describing a software architecture. Furthermore, practical application of the technique in an industrial research project will be discussed.

1 http://www.danfoss.com/
2 http://www.systematic.dk/
3 http://www.isis.alexandra.dk/
In the following, we assume that the reader has some knowledge of the UML metamodel [19].

2 Using UML at a Software Architectural Level

The software architecture of a system may be defined as

the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them [1, p. 21]

From a software safety point of view there are a number of reasons for focusing on software architecture:

- A software architecture design is software design at an early stage and at a high level of abstraction. The design embodied in a software architecture can often be analyzed in more manageable ways than a complete system description and the software architecture is crucial in obtaining desired system qualities [1,4].
- A software architecture can be communicated among different stakeholders. Having system descriptions on a high level of abstraction enables sharing of these descriptions among, e.g., system engineers, software developers, customers, and end users which is particularly important in safety-critical system development since a majority of problems here stem from requirements problems [13].

In order to handle the complexity of software systems, software architectures are preferably described as different structures or from different viewpoints [11,21]. Recent approaches to software architecture description advocate having three overall viewpoints [3]:

- Module viewpoint. This viewpoint is concerned with how functionality of the system maps to static implementation units such as classes and packages.
- Component-and-Connector (C&C) viewpoint. This viewpoint is concerned with the runtime mapping of functionality to components and connectors of the architecture.
- Allocation viewpoint. This viewpoint is concerned with how software entities are mapped to environmental entities.

This is combined with a fourth central architectural requirements viewpoint – akin to Kruchten’s +1 view [11] – in which, e.g., quality attribute criteria, requirements, and use cases of architectural significance, which cannot easily be described in a specific viewpoint, are described. Our approach to software architecture description follows this closely. This effectively restricts the number of UML modeling elements used to a subset of interest for a large number of models. Section 4 considers these in the context of HAZOP analyses.

3 HAZOP for UML Elements

This section presents general ideas and recommendations about how to perform HAZOP studies on UML models. Section 3.1 provides a general introduction to HAZOP studies. Section 3.2 introduces a systematic approach for identifying the elements of UML models that should be considered in a HAZOP study.
3.1 Introduction to HAZOP

A HAZOP study is a systematic investigation of design representations which are conventional, descriptive models of a system’s design [15]. The purpose of a HAZOP study is to identify all possible deviations from the design’s expected operation and all hazards associated with these deviations [13], and to avoid continuing development of designs with potential hazards [15]. The technique uses guide words to promote creative thinking about the ways in which hazardous situations might occur. A guide word is used to express a particular kind of deviation. For example, applying the guide words before and after to the timing of a message in a sequence diagram will encourage designers to consider whether or not hazards can occur if the message is sent earlier or later within the sequence of messages.

Table 1 shows the generic list of guide words and interpretations from the standard for HAZOP studies for programmable electronics [15]. The generic guide word interpretations can be used for various software design representations. However, it is recommended that diagram-specific interpretations and guide words be developed when necessary. Not all guide words will be appropriate for all system design representations.

<table>
<thead>
<tr>
<th>Guide Word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>This is the complete negation of the design intention. No part of the intention is achieved and nothing else happens.</td>
</tr>
<tr>
<td>More</td>
<td>This is a quantitative increase.</td>
</tr>
<tr>
<td>Less</td>
<td>This is a quantitative decrease.</td>
</tr>
<tr>
<td>As well as</td>
<td>All the design intention is achieved together with additions.</td>
</tr>
<tr>
<td>Part of</td>
<td>Only some of the design intention is achieved.</td>
</tr>
<tr>
<td>Reverse</td>
<td>The logical opposite of the intention is achieved.</td>
</tr>
<tr>
<td>Other than</td>
<td>Complete substitution, where no part of the original intention is achieved but something quite different happens.</td>
</tr>
<tr>
<td>Early</td>
<td>Something happens earlier than expected relative to clock time.</td>
</tr>
<tr>
<td>Late</td>
<td>Something happens later than expected relative to clock time.</td>
</tr>
<tr>
<td>Before</td>
<td>Something happens before it is expected, relating to order or sequence.</td>
</tr>
<tr>
<td>After</td>
<td>Something happens after it is expected, relating to order or sequence.</td>
</tr>
</tbody>
</table>

Table 1. Generic guide words from [15].

HAZOP studies are based on examining design representations of a system. The recommended steps in a HAZOP study are to:

1. Identify each entity in the design representation
2. Identify attributes, i.e. physical or logical properties, for each entity
3. Investigate deviations from design intent by applying guide words to attributes
4. Investigate, for each deviation, the causes and consequences

In the following, the term HAZOP attributes will refer to the physical and logical properties of an entity in a design representation. The terms attributes and UML attributes will refer to named pieces of the declared state in UML classifiers [19].

3.2 HAZOP for UML Models

Any UML model element can be considered as a HAZOP entity, and its UML attributes, composed elements, and associated elements, as defined by the UML metamodel, as its
HAZOP attributes. An example of this would be to consider a UML Class as a HAZOP entity and to consider Class.isActive its Attributes, Operations, and Methods as HAZOP attributes.

Given that this may be done recursively, one ends up with a prohibitively large number of HAZOP attributes and corresponding guide words which need to be used in a HAZOP analysis. Thus, our approach is to base the HAZOP analysis on an architectural description, which limits the number of UML elements to be considered and which also contains UML diagrams from which to start the analysis. Figure 1 shows the central concepts from which to start a HAZOP analysis in this approach. The left part of Figure 1 shows the concepts which are specific to our approach to architectural description (which is conformant with IEEE Std 1471-2000 [21]) and confines the number of UML elements to eventually consider. The right part of Figure 1 shows classes specific to the UML metamodel. Diagram and DiagramElement are a simplified subset of the UML diagram interchange metamodel [18] which is connected to the UML ModelElement metaclass [19]. As an example of part of the metamodel, viz., Namespace, its subclass Package, and its composition Namespace.ownedElements of ModelElements is shown.

We may then consider each class in Figure 1 from left to right as a HAZOP entity and its composed elements as HAZOP attributes. For each UML metamodel element, its attributes and associations of metamodel elements may also be considered as HAZOP attributes. Moreover, the elements we initially consider as UML-based HAZOP attributes are the UML model elements shown on diagrams.

In the example in Figure 1, this would lead us to consider in turn, Architectural Description, subclasses of View, Diagrams, Packages, and their composed ModelElements as HAZOP entities. Moreover, all of these, except Architectural Descriptions, may be considered HAZOP attributes in themselves. An example of this would be Module View which is a HAZOP attribute of Architectural Description as well as a HAZOP entity having Package and Class diagrams as HAZOP attributes. Section 4 contains suggested
guide words and interpretations for several different kinds of UML diagrams and associated model elements.

We are primarily considering UML elements at a “diagram level”. While relationships, e.g. Associations and Links, can be drawn simply in a UML diagram, their definitions in the UML metamodel are more complex. For example, a binary Association is composed of two AssociationEnds, each of which is associated with a Classifier, such as a Class. This relatively complex relationship is drawn as a solid line between two Classifiers. Working on a diagram level, if Association exists, it should be implicitly understood that the corresponding AssociationEnds also exist, and similarly for other relationships.

4 HAZOP for UML-Based Software Architecture Descriptions

This section identifies HAZOP entities, HAZOP attributes, and guide word interpretations for UML-based software architecture descriptions that conform to the description in Sections 2 and 3.2. HAZOP cannot be applied to a design representation if the design intention of the design representation and the entities that it contains is unknown. For example, it would be meaningless to try to apply HAZOP to a class diagram if it not known whether the class diagram should show all classes for the software system, or if it should show the classes in a small subsystem.

4.1 Guide Words for Composed Elements

Composition is a central modeling mechanism in the UML metamodel. Rather than defining specific guide word interpretations for all of the different kinds of composed metamodel elements that can be identified from a metamodel element, Table 2 shows guide words and their interpretations for generic composed elements. These guide word can be used to investigate whether or not an entity contains all of the composed elements necessary for achieving the design intention of the entity within the context of a particular design representation. As an example, the guide words in Table 2 can be used to consider whether or not existing classes and associations deviate from the design intention of a class diagram.

<table>
<thead>
<tr>
<th>Composing Entity</th>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composed elements</td>
<td>None</td>
<td>The entity contains none of the necessary composed elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>As well as</td>
<td>The entity contains all of the necessary composed elements, as well as additional composed elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Part of</td>
<td>The entity contains some, but not all, of the necessary composed elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other than</td>
<td>The entity contains composed elements, but the composed elements do not fulfill the design intention</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Suggested guide word interpretations for composed and associated elements of an entity.
4.2 Architecture Description

In our approach, the most abstract entity to which HAZOP analysis can be applied is the Architecture Description itself. Table 3 shows HAZOP attributes and guide words that have been identified for the architectural description described in Section 2.

<table>
<thead>
<tr>
<th>Entity=Architectural Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute</strong></td>
</tr>
<tr>
<td>Module View</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C&amp;C View</td>
</tr>
<tr>
<td>Allocation View</td>
</tr>
<tr>
<td>Arch. Req. View</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 3. Suggested guide word interpretations for attributes of architectural descriptions.

4.3 Module View

One step in performing HAZOP analysis on the Module View, is to consider the view itself as a HAZOP entity. HAZOP attributes of the Module View are class diagrams and package diagrams, and guide words for these HAZOP attributes can be derived from Table 2.

**Class Diagrams** The HAZOP attributes of a class diagram are its composed elements, i.e. classes, associations, generalizations, dependencies, packages, interfaces, objects, and links. Guide words for these HAZOP attributes can be extrapolated from Table 2. When a class diagram is considered to be an entity, HAZOP analysis should not be concerned with lower-level details such as class operations or association multiplicities. These details should be considered if the class or association is the entity that is the focus of a HAZOP analysis.

In principle, when applying guide words to associations, dependencies and generalizations, one must consider whether or not the relationship should be defined for each relevant group$^4$ of model elements in the class diagram, in order to determine whether or not the necessary relationships have been included in the class diagram. This may seem unnecessarily complicated, however, it may help to identify associations, dependencies, and generalizations that could have significant impact on safety-critical aspects of the system.

$^4$ Some relationships, such as associations, can have more than two endpoints, which is why it is necessary to consider groups of model elements.
**Classes** Each of the elements in a class diagram can also be considered as an entity during HAZOP analysis. Table 4 shows HAZOP attributes and guide words for individual classes in a class diagram.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isActive</td>
<td>Reverse</td>
<td>The class is active, even though it should not be. Or the class is not active, even though it ought to be.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Attributes</td>
<td>See Table 2</td>
</tr>
<tr>
<td>Operations</td>
<td>See Table 2</td>
</tr>
</tbody>
</table>

**Table 4.** Suggested guide word interpretations for attributes of Class.

The design intention of a particular class diagram will dictate whether or not it is important to consider UML attributes and operations during the HAZOP analysis of the software architecture. More precise interpretations for guide words for UML attributes and operations can be explicitly defined as necessary. Methods contain low-level implementation details, and will therefore generally not be found in architecture descriptions. HAZOP analysis of a subclass may have to (re)consider the attributes of all of the superclasses of the class.

**Associations** Table 5 shows suggested HAZOP attributes and guide words for associations. Each of the HAZOP attributes in Table 5 must be considered for each of the endpoints of an association and not for the association itself.

The UML attribute *aggregation* for AssociationEnds determine the aggregation kind of an association, i.e., *none*, *composition*, or *aggregation*. The UML well-formedness rules dictate that certain conditions must hold for associations. For example, at most one endpoint of an association can be a composition. These rules will automatically be fulfilled if mature UML tools are used to create UML diagrams. In such cases, it is not necessary to apply guide words to the aggregation attribute of the other endpoint of the composition relationship. Similar guidelines can be defined in order to reduce the scope of a HAZOP analysis of UML diagrams.

The changeability attribute of an association determines whether links between objects are *changeable*, *frozen* or *addOnly* at runtime. Associations are changeable per default. This attribute could prove to be quite useful when developing safety-critical software. By requiring that an association is frozen, i.e. that instances of the association may not be destroyed, it may be very easy to argue that certain safety requirements can be fulfilled. e.g. that a safety-critical object is always accessible because links to it can never be destroyed.

**Package Diagrams and Packages** We define a *package diagram* to be a class diagram that only contains packages, and relationships between packages. This kind of diagrams are useful for showing the decomposition structure of a software design. HAZOP attributes for a package diagram are its composed elements, i.e. packages, generalizations, and dependencies. Guide words for these HAZOP attributes are shown in Table 2.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNavigable</td>
<td>Reverse</td>
<td>The association is navigable to the endpoint in question, but it ought not to be. Or the association is not navigable to the endpoint in question, but it ought to be.</td>
</tr>
<tr>
<td>aggregation</td>
<td>Other than</td>
<td>The aggregation kind of the endpoint is not what is intended.</td>
</tr>
<tr>
<td>changeability</td>
<td>Other than</td>
<td>The changeability of the association endpoint is not what is intended.</td>
</tr>
<tr>
<td>multiplicity</td>
<td>None</td>
<td>No multiplicity is specified, despite the fact that a particular multiplicity should have been specified.</td>
</tr>
<tr>
<td>None</td>
<td>More/Less</td>
<td>The multiplicity set consists of only one value that is higher/lower than intended.</td>
</tr>
<tr>
<td>None</td>
<td>As well as</td>
<td>The multiplicity set contains all of the intended integers, as well as additional values.</td>
</tr>
<tr>
<td>None</td>
<td>Part of</td>
<td>The multiplicity set contains only some of the intended integers.</td>
</tr>
<tr>
<td>None</td>
<td>Other than</td>
<td>The multiplicity set contains none of the intended values, but it does contain other values.</td>
</tr>
</tbody>
</table>

Table 5. Suggested guide word interpretations for attributes of associations.

A package is composed of its ownedElements, which are, e.g., packages, classes, associations, and dependencies [19]. Guide words for the ownedElements of a package can be derived from Table 2.

### 4.4 Component and Connector View

When the C&C view is considered as a HAZOP entity, its HAZOP attributes are object diagrams and sequence diagrams. Guide words for these HAZOP attributes can be derived from Table 2.

**Object Diagrams, Objects, and Links** An object diagram provides a snapshot of detailed state of the system at a point in time [19]. At an architectural level, this can be used to communicate and design the overall runtime functional decomposition of the system. Object diagrams are composed of objects, links, and data values, which are the three HAZOP attributes of object diagrams. Guide words and their interpretations for these attributes can be found in Table 2.

**Sequence Diagrams** A sequence diagram is used to represent a use case or scenario. Lano et al [12] presents a set of guide words for elements in sequence diagrams (in their case “objects” and “messages”) which is not tied to the UML metamodel per se. Sequence diagrams are typically composed of Messages and ClassifierRoles, and guide words for these HAZOP attributes can be derived from Table 2. Tables 6 and 7 present guide words for HAZOP attributes for ClassifierRoles and Messages, respectively. Some of the guide word interpretations are derived from [12].
### Table 6. Suggested guide word interpretations for attributes of ClassifierRoles.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>No</td>
<td>Wrong base Classifier is specified.</td>
</tr>
<tr>
<td></td>
<td>Reverse</td>
<td>A safety-related Classifier is represented rather than a non-safety-related Classifier (or vice versa).</td>
</tr>
<tr>
<td>multiplicity</td>
<td>More</td>
<td>More elements than intended are represented.</td>
</tr>
<tr>
<td></td>
<td>Less</td>
<td>Less elements than intended are represented.</td>
</tr>
</tbody>
</table>

### Table 7. Suggested guide word interpretations for attributes of Messages.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor/ successor</td>
<td>No</td>
<td>Message not sent when intended (to any destination).</td>
</tr>
<tr>
<td></td>
<td>Other than</td>
<td>Message sent to wrong object.</td>
</tr>
<tr>
<td></td>
<td>As well as</td>
<td>Message sent to correct object and also an incorrect object.</td>
</tr>
<tr>
<td></td>
<td>Sooner</td>
<td>Source and destination objects are reversed.</td>
</tr>
<tr>
<td></td>
<td>Later</td>
<td>Message sent to more objects than intended.</td>
</tr>
<tr>
<td></td>
<td>More</td>
<td>Message sent to fewer objects than intended.</td>
</tr>
<tr>
<td></td>
<td>Less</td>
<td>Message sent to more objects than intended.</td>
</tr>
</tbody>
</table>
4.5 Allocation View

The major diagram type of the allocation view is the UML deployment diagram which shows how software components and objects map to deployment units.

Deployment Diagram Deployment diagrams are composed of nodes, components and links, and guide words for these HAZOP attributes can be derived from Table 2. Since nodes are composed of components, and components are composed of objects, Table 2 can be used to derive guide words when nodes and components are considered as entities in a HAZOP study. HAZOP attributes and guide word interpretations for links are shown in Table 8.

<table>
<thead>
<tr>
<th>Entity=Link</th>
<th>Attribute</th>
<th>Guide word</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Less</td>
<td>A protocol not providing the necessary resources (e.g., throughput, latency, or delivery guarantees) is used.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>As well as</td>
<td>A communication protocol more than fulfilling the design intention is used (e.g., using USB over RS232 focusing only on throughput).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Part of</td>
<td>A communication protocol only fulfilling part of the design intention is used.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other than</td>
<td>Communication protocol which does not fulfill the design intention is used (e.g., using IEEE 802.11 for isochronous communication).</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Suggested guide word interpretations for attributes of Links.

4.6 Architectural Requirements View

The main UML artifact for requirements are use cases and use case diagrams are thus the only diagram type which is considered in this section. A typical requirement description vehicle for safety critical systems is statecharts which are not considered here since this is considered in detail in [12] and [15]. Statecharts are also often used in the C&C view.

Use Case Diagrams Use case diagrams describe the structure of requirements by use cases. The primary representation of use cases is most often a textual description (e.g., in the format of [5]), but as an auxiliary artefact, use case diagrams may provide overview, and are, moreover, amenable to HAZOP analysis. The primary HAZOP analysis of use case diagrams is at the level of diagrams.

5 Preliminary Experience Report

The proposed HAZOP method was applied during the preliminary stages of the development of a UML-based software architecture for a frequency converter with integrated safety functions at Danfoss Drives A/S. Frequency converters are used to control the speed of motors. Such frequency converters are increasingly applied in the process industry and production, where they replace classical electromechanical safety devices such as power relays.
<table>
<thead>
<tr>
<th>Entity = Use Case Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
</tr>
<tr>
<td>Actors</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Use Cases</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Include Relationships</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 9. Suggested guide word interpretations for attributes of Use Case Diagrams.

5.1 Frequency Converter

**Hardware.** Figure 2 shows the hardware structure of the frequency inverter. The two blocks *PWM Generator* and *Power Electronics* make up the normal, “non safety-related” part of a frequency converter, and they transform the 50Hz main supply to the output (*u*, *v*, *w*).

The safety functionality is achieved by an additional subsystem composed of *Channels 1* and 2, each consisting of a microprocessor (*uP*), a switch-off path (*Switch off*), a number of *Digital Inputs* and one *Speed Information* input. The two microprocessors can, independently from each other, activate its own switch-off path to stop the rotational torque in the motor. The two *Channels* cross-monitor each other through *Feedbacks 1* and 2 and through the *Cross Communication* connection. Channel 1 is also connected to a PROFIbus® fieldbus with fail-safe properties (PROFIsafe®).

**Frequency Converter Software.** A number of safety functions can be realized on the basis of the physical capabilities of the frequency converter. The logic of these safety functions is implemented in software that runs on the two microprocessors. A specific safety function is triggered upon reception of signals either at digital inputs at each of the *Channels*, or via the fieldbus connection at *Channel 1*.

The simplest safety function is a so-called ‘uncontrolled stop’ which immediately stops torque generation in the motor. Another safety function is a ‘controlled stop’, where the stop of torque generation is delayed, allowing the non-safety-related part of the frequency converter to ramp the motor down in a controlled way. A more complex example is the ‘safe speed’ where an uncontrolled stop is made if the motor speed is above a set limit.

All diagnostic functionality with respect to cross monitoring and self monitoring of the *Channels* is implemented in software. On detection of a dangerous failure, an appropriate fault reaction is initiated.

5.2 HAZOP Analysis of Safe Inverter Software Architecture

In a design of a safety-critical system HAZOP analysis would typically take place throughout the development process in a series of workshops in which potential hazards are
recorded in order to be investigated further. During a HAZOP study guide words are systematically applied to attributes of entities in design representations, as described in Section 3.1 and [15]. HAZOP analysis of software architecture should lead to early identification of critical behaviour that can be eliminated, alleviated, or flagged for in-depth analysis in later stages.

The method for HAZOP and UML proposed in this paper has been used to identify critical behavior in a preliminary version of the software architecture for the frequency converter. The results shown in this section were among others obtained during informal workshops. In this case, a few representative entities were selected from different design representations, and guide words were applied to the attributes of these entities. Results of the analysis are recorded in tables that indicate causes, consequences, and recommended actions for the problems that are identified. The analysis takes its outset in the first three viewpoints of the software architecture, and we here present an example for each of these views.

**Module Viewpoint** Figure 3 shows the overall module decomposition of the frequency inverter software in a central Control subsystem and among other subsystems which im-
plement the safety functions (SafetyFunctions) and regular diagnosis of the consistency of the two Channels of the frequency converter (Diagnosis). The External package contains classes which are responsible for implementing communication with hardware and devices external to the microprocessors. In order to improve the readability of Fig. 3, the owned elements of the five main packages are not shown.

Table 10 shows examples of the results of applying guide words to entities in this diagram. HAZOP analysis of the Module Viewpoint has identified ways in which the current architecture could be improved in order to avoid hazards, e.g. by reducing the number of unnecessary superclasses and by replacing associations to abstract classes with associations to non-abstract classes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies (Entity = Fig. 3)</td>
<td>As well as</td>
<td>Other packages than Control depend on Diagnosis. This may lead to dangerous circular dependencies since a design intention is that diagnoses run in the background, the run being independent of other parts of the system.</td>
</tr>
<tr>
<td>ownedElements (Entity = Diagnosis package)</td>
<td>As well as</td>
<td>The inheritance relationships of the ownedElements is complicated. This may result in dead code, or code that is too complicated to verify, and the final program may require too much space. Recommendation: more detailed analysis of ownedElements which may lead to revised design or evidence that the design is acceptable.</td>
</tr>
<tr>
<td>associations (Entity = Controller class in Control package)</td>
<td>Part of</td>
<td>Several associations point to abstract classes in other packages. This may result in associations to too many, too few, or the wrong kinds of objects at runtime. Recommendation: specify associations to non-abstract classes whenever possible.</td>
</tr>
</tbody>
</table>

Table 10. Part of HAZOP analysis of entities in Figure 3.
C&C Viewpoint  Figure 4 shows the overall flow of control in the design of one of the central use cases related to the safe inverter, viz., the invocation of a safety function; in this case the controlled stop (SafeDelay in the diagram) safety function. Table 11 shows examples of the results of applying guide words to entities in this diagram. HAZOP analysis of the C&C Viewpoint has identified ambiguities and oversights in sequence diagrams for safety-critical scenarios, and has resulted in recommendations for design revisions.

Allocation Viewpoint  Figure 5 shows the major nodes of the frequency inverter from a software point of view (compare Figure 2). Components and objects are not shown. Table 12 shows examples of the results of applying guide words to entities in this diagram.

5.3 Discussion
A number of design problems and potential hazards were identified for the preliminary software architecture description of the frequency converter, despite the fact that a rigorous and thorough HAZOP analysis of the entire architecture description has not been performed. The results of this analysis will be used to improve the architecture description before it is used as the basis for detailed design of the software.
### Table 11. Part of HAZOP analysis of entities in Figure 4.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassiferRoles (Entity = Fig. 4)</td>
<td>Part of</td>
<td>Some needed ClassiferRoles are not present. Hazards could occur if information regarding the start and/or stop of a safety function is not sent to all of the necessary output devices. Recommendation: check whether or not it is necessary to send information to additional output devices, such as the PC in Fig. 5.</td>
</tr>
<tr>
<td>Messages (Entity = Fig. 4)</td>
<td>Part of</td>
<td>The message ProfSafe.checkDSFRequest is not sent often enough. It must be sent before each time SafeDelay.run is sent to ensure that the request for the safety function has not been removed. Recommendation: update the sequence diagram.</td>
</tr>
<tr>
<td>successor (Entity = SafeDelay calling Can.sendMessage)</td>
<td>Later</td>
<td>It may be problematic that SafeDelay.checkDelayElapsed is called only after a message has been sent to externals and the safety function has started depending on the actual delay in the controlled stop and the latency in sending messages.</td>
</tr>
</tbody>
</table>

### Table 12. Part of HAZOP analysis of entities in Figure 5.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Guide word</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>name (Entity = link between uP2 and uP1)</td>
<td>Less</td>
<td>It is intended that communication between the two microprocessors should use dual-ported RAM. It is critical that the RAM can hold a sufficient number of concurrent messages, and this must be checked before developing this architecture further.</td>
</tr>
<tr>
<td>name (Entity = link between uP2 and PLC)</td>
<td>Other than</td>
<td>It is not specified that the communication protocol is safety-critical (and thus uses the PROFIsafe® protocol on top of PROFIbus®). In the specific diagram, low level hardware interfaces or protocols are, however, generally shown.</td>
</tr>
</tbody>
</table>
Providing tool support for this approach would clearly be advantageous. It should be straightforward to implement a tool that could, e.g., propose HAZOP attributes and guide words for UML elements; such a tool could either be a plug-in to a UML tool or a stand-alone tool. Developing tool support for this approach is beyond the scope of this project, but Lano et al [12] do mention a tool for HAZOP analysis of class diagrams.

As mentioned previously, a full HAZOP analysis of a detailed object-oriented design model in UML would be very resource consuming. The architecture description for the frequency inverter – which Section 5.2 gives examples of – currently consists of fewer than 15 UML diagrams of which more than half are sequence diagrams.

Even at this level of design, it may be too much to expect that HAZOP analysis will be performed for all UML elements. One approach to limiting the number of sequence diagrams to consider would be to choose a representative or a particularly critical set of scenarios to consider. In the concrete case, it is essential that safety functions can be requested and that most of them must be able to disable the signal to the motor. On the other hand, other activities, such as diagnosis are not (as) safety critical. For the static diagrams, the HAZOP analysis is much more manageable, however, and full HAZOP may be viable in the context of safety-critical system development.

An alternative approach combining Fault Tree analysis and HAZOP analysis can also be used to reduce the number of UML elements that must be considered when examining UML diagrams. In this approach fault trees based on the UML diagrams are created, and only the UML elements, such as classes and relationships, that are not covered by the fault trees should be considered as entities during HAZOP analysis.
A premise of the work presented in this paper is that system qualities related to software safety are to a significant extent determined by the software architecture of the system. This is supported by current practice and research [1], but further research is needed to determine to which extent this is the case. In particular, it would be of interest to know how many and which types of hazards would typically be found at the level of software architecture.

6 Summary

This paper has introduced a systematic approach to applying Hazard and Operability (HAZOP) analyses to Unified Modeling Language (UML) models of safety-critical systems. HAZOP takes as its outset design intentions and design representation of those intentions. By using guide words applied to parts of these representations, developers systematically investigate the design in order to discover potential hazards. If UML is used for design representation, a clear way of considering guide word interpretations for parts of UML elements is needed.

We iteratively consider UML elements as HAZOP entities for whose composed elements, attributes, and associated elements guide words may be applied in a HAZOP analysis. To limit the number of UML elements to be considered in a HAZOP analysis, we take as outset a high-level UML-based architectural description.

Furthermore, we have illustrated how the approach has been used during the preliminary development stages of object-oriented software for a safety-critical frequency converter. The approach represents a step on the way of reconciling object-orientation and the development of safety-critical systems.

Acknowledgements

The work reported in this paper has been supported by ISIS Katrinebjerg Software5.

References


5 http://www.isis.alexandra.dk/software


Factors Determining Effective Realization of MDA in Industry\textsuperscript{1}

Miroslaw Staron*, Ludwik Kuzniarz*, Ludwik Wallin**

*Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
Box 520, Soft Center
SE-372 25 Ronneby, Sweden

**Volvo Information Technology
Göteborg, Sweden

Abstract. The Model Driven Architecture (MDA) initiative gains a considerable and growing amount of interest in both academia and industry. It seeks to raise a level of abstraction while creating models by using distinct models for presenting platform dependent and platform specific information. MDA core ideas are also aimed to improve the model creation process by automating transformations between models. Automatic transformations between models and using models for different purposes require using more precise languages for the creation of models. The Unified Modeling Language (UML) is placed in the center of MDA and its customization is an essential aspect of the successful realization of MDA. The success is determined by certain factors. In this paper, we identify key factors for the efficient accomplishment of the MDA. It is done by means of industrial case study. The case study is performed in a context of a large IT company with distributed development unit developing complex systems. The factors identified are grouped into two categories – associated with usage and development of an MDA-based framework. The factors are important for the success in MDA realization endeavors.

1 Introduction

The essential idea behind Model Driven Architecture (MDA, [1]) approach to software development is the separation between the specification of the systems essential functionality and the implementation of these systems using specific implementation platforms. The Unified Modeling Language (UML, [2]) is becoming widely accepted standard notation for expressing artefacts within software development process and is the core modeling language of MDA. Since models used and produced in the development process are to be semantically consistent abstractions of a system, they represent views of a system within a development process made on different levels of abstraction and from different perspectives. A basic activity performed within such development process is producing new or modifying existing models. This activity can be viewed as performing model transformations. In the case of MDA the most significant transformations are transformations from platform independent to more platform specific models and finally producing a running code in a target environment. These leads to a vision on MDA as a set of automatic transformations performed on

\textsuperscript{1} This work was partially funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).
models aiming at increasing level of details [1, 3, 4]. Nevertheless, it is often impossible to express the details of models with the standard UML modeling constructs. Therefore, the modeling language has to be customized to suit the purposes of each model. Such customizations can be done with the use of existing extension mechanisms of profiles and stereotypes. Using different profiles for modeling platform specific and platform independent models requires a definition of a mapping between the profiles. The mapping is usually defined as part of the software development process and quite often it can be automated. Making the transformation automated requires some initial effort of precise specification of the transformations and their later implementation. There can be many gains in terms of time and effort, when the transformations are used. Several factors influence the effectiveness of the MDA realization process. The paper analyzes a case of industrial project, in which the MDA principles were envisioned. The case study resulted in identification of the core factors determining the effective realization of the principles of model driven development.

The paper begins with a short description of the Model Driven Architecture, its principles and purposes, followed by a description of the case study performed on its industrial realization. The subsequent section presents results from the case study – factors determining the effective envisioning of MDA. Finally, the analysis of validity of results and conclusions are presented.

2 Model Driven Architecture

The Model Driven Architecture (MDA, [1]) describes a set of standards useful in model creation in model driven software development and a set of guidelines for software development processes [5]. The main principle of MDA is to raise the level of abstraction of models and to separate the essential system functionality from implementation details. The Platform Independent Models (PIMs) creation principles encourage omission of implementation specific details, favoring modeling of software core functionality. They provide formal specifications of the structure and function of the system that abstract away technical details [1], which is especially important from the perspective of integration between platforms [6]. On the other hand, Platform Specific Models (PSMs) creation principles encourage adding the implementation specific information to models, at the same time prohibiting changing the systems platform independent functionality. The software development based on MDA should include series of transformations from PIMs to PSMs – preferably automatic. In order to improve such aspects of software development as reusability, traceability and consistency, the transformations should encapsulate activities that are commonly repeated. However, to be able to perform automatic transformations, some assistance is required from model developers. It is them who should decide which are the implementation platforms and which elements should be transformed by which transformation.

The separation of concerns principle in MDA requires that PIMs and PSMs are expressed using domain specific and implementation oriented modeling languages. These languages should provide different features and modeling possibilities. The PIM specific modeling language should be as general as possible at the same time restricting the set of models possible to be built with it. The restriction should forbid creating of models, which cannot be transformed into platform specific models – i.e. constructs that are not possible to implement should not be possible to model. The PSM specific language should be as detailed as possible allowing expressing all elements that can be implemented. The PSMs are the basis for code generation.
One PIM should be the basis of creation of multiple PSMs, which are targeted towards specific platforms – for example Java or .NET’s C#. The PSM should be as detailed as possible to provide possibly full code generation from the model to the specific programming environment. As it is presented in Fig. 1, each PSM should be the basis of code generation for the specific platform.

2.1 Role of the Unified Modeling Language and its customization

The Unified Modeling Language (UML, [2]) is the core language for model creation in MDA. According to the MDA specification, the usage of UML can help in raising the level of abstraction of models – from the previously used textual IDL (Interface Definition Language) – or alike – descriptions to the graphical UML models. Using UML allows expressing invariants and restrictions with dedicated constraints languages. One of such languages is the Object Constraint Language (OCL, [7]), defined as part of the UML specification. The customization mechanisms of UML allow tuning this general purpose language towards the specific needs of PIMs and PSMs. The mechanisms can be based on the notion of a profile, which is a set of stereotypes. Stereotypes are means of extending the base modeling element’s semantics so that it suits a specific purpose. Stereotypes can also add additional properties – information – to the stereotyped element, just as the meta-classes in the UML specification – UML meta-model – define standard properties of model elements. Constraints (usually defined in OCL) are attached to stereotypes, thus allowing defining restrictions on the usage of the base modeling element. They are especially useful in restricting the set of models that are possible to be built with the given profile. In Fig. 1, each profile defines a separate language specific for its domain, such as the PIM, or PSMs. It is depicted by a distinct pattern of the background of the model.

An alternative to the profile mechanism is a technique of meta-modeling. The technique is based on the assumption that the UML specification (UML meta-model) is changed, which in fact is a creation of a new – UML-like language [8-11]. Although it is a much more powerful technique, it is still not widely used in industry because of the lack of support in UML tools – a factor that is particularly important in the industrial context. It should not be expected that UML tools have meta-modeling capability, but rather that there exist tools which provide this capability and later provide a means of integration of their output with UML tools. This reflects two different roles that are present in the
MDA realization endeavors – the creators of the language extensions and the developers using the extensions.

2.2 Role of model transformations

The process of creation of models according to MDA principles should be repeatable and should include the creation of PIM, its transformation to PSM and then code generation from the PSM in an automatic way [12]. Multiple incremental iterations should be possible in such processes. As mentioned before, model transformations should aid in updating models after transformation in case the model that was being transformed was updated. The situation can occur when the PIM model has been changed in one of the iterations, and the PSM model should be updated. However, they cannot be re-created because they already contain some additional – platform specific information that was added manually. Such an approach leads to important aspects of model consistency with the refinement relationships between models. As the MDA documentation indicates the refinements should be aimed at making the models more detailed. Therefore it is important that the models at different abstraction level are consistent with each other.

Automatic model transformations aid developers, thus improving their productivity and quality of the produced code [12, 13]. However, to be able to automatically transform models, a lot of additional information about the models is required. In addition to the data provided by elements in the input model certain transformation specific information is required (for example the information about the target data model in the database modeling process). Such information can be added to the input model elements by attaching a stereotype to it, which contains the information in form of a set of tagged values. In UML 1.x family it was possible to use other mechanisms – such as tag definitions which are not part of a stereotype. However, in the forthcoming UML 2.0 specifcation {Object Management Group, 2003 #1414}, such constructs are not present so they are not considered in the paper. It so happened that the studied case did not use such constructs.

3 Case Study Design

The case study was performed in the large-size IT company with world-wide distributed offices – Volvo Information Technology (Volvo IT). The placement of the project development team is in the Volvo IT headquarters in Gothenburg, Sweden, and it was where the study was performed. The aim of the study was to identify the main factors that affect the effective MDA realization in the JNX (Java & .NET & XDE) project.

3.1 Context

Volvo IT is a wholly-owned subsidiary of AB Volvo, one of the largest industrial groups in the Nordic Region. Volvo IT provides all types of industrial IT solutions in a variety of technical environments. The company was formed in 1998, through a merger of all IT resources from different Volvo Group companies. Volvo IT provides Volvo Group, Volvo Cars (since 1999 owned by the Ford Motor Company) and other selected customers with cost-effective IT-solutions resulting in long-term business value.

Volvo IT is a comprehensive global IT company with about 4800 personnel and annual sales in excess of SEK 5 billion.

The Application Development Techniques organization, which is represented at the larger sites, is responsible for supporting application development and maintenance teams with development processes, methods, tools and application development environments.

The studied project of the Application Development Techniques was aimed at providing a common framework for software development across the company. The rationale behind introducing the project was to improve the quality of the generated code, to raise the abstraction level of models, to shorten development time and provide a
common process framework. The aim of the project was to provide a set of common mechanisms for software development which hide the tasks that could be performed automatically. It allows the developers to focus on business functionality, instead of tediously repeating the common implementation-related tasks. The business models in the case of the company are platform independent models (PIMs), which specify the systems’ core functionality, abstracting from any implementation details. Such models are the basis for creation of the prescriptive platform specific models (PSMs), which specify how the business solution is realized by the components in the target platform.

The creation of the models is done in an automatic way via model transformations. Both PIMs and PSMs are described using a customized UML version. The customization is achieved by using the profile mechanism and extensive use of stereotypes, tagged values and constraints.

The study was performed during half of the lifecycle of the JNX project – from its middle stage to the completion of the project. The project was performed in an iterative and incremental way. It was the second (and final) iteration of the solution when the study was done. The second iteration was based on the previous – limited version – of the solution providing only the basic (core JNX) functionality. The core reusable functionality of the JNX framework includes mechanisms for:

- persistency,
- logging,
- transactions,
- error handling,
- configuration,
- remote communication, and messaging

The core mechanisms are provided via a black-box reuse mechanism in the code and they are automatically included into the solution with help of automatic model transformations. The functionality is added to all software solutions created with the help of JNX and it is transparent to the developer who uses the JNX framework. The developer’s role is limited to using the provided mechanisms in the created solutions. An overview of the JNX framework is presented in Fig. 2, which shows in detail a single path of modeling as presented in Fig. 1. Solid arrow lines in the figure represent automatic model transformations and the open arrows represent manual transformations. The two profiles used in the solution are presented as different background of the models rectangles.

![Fig. 2. Overview of the JNX framework](image)

The set of automatic model transformations developed in the framework consists of four kinds of transformations:

- PIM to PIM marked,
- PIM marked to PSM,
- PSM to PSM, and
- PSM to source code

The transformations between two PIMs – PIM refinements – allow application of domain specific (and platform independent) design patterns thus preparing the models so that they contain the required information for transformation from PIM to PSM –
updating the PIM model with additional information. The transformations from PIM to PSM create the initial version of the PSM model or provide a means of updating the existing PSM model based on the changes introduced in the PIM model.

The transformations between two PSMs – PSM refinements – add certain information to the model elements based on the structure of the model. They add some additional information to the PSM model for complete code generation for the model. The information contains model-specific properties of model elements that is required to produce full source code. The code generation transformations produce source code of the PSM classes in the desired implementation language and full source code for:

- persistence management in relational databases for simple and composite objects,
- message transformation between flat, XML-based, object formats, and
- template methods for additional business rules

The transformations in the JNX framework provide a means of hiding the implementation specific details, which are now transparent for developers using the framework.

Performing the automatic transformations requires certain additional information to be added to the standard modeling elements in UML. Both the PIM models and the PSM models must contain domain specific information that is required for the transformation. The data is added in forms of tagged values defined as part of stereotypes enclosed in profiles. There are two different profiles – sets of stereotypes – for the PIM (gray crossed background of model in Fig. 2.) and PSM models (dotted background of model in Fig. 2.). The profiles for PIM marked were created as a result of requirements posed by transformations creators. The profiles contain information required to perform the transformations. The information is also used in PSM as a means of synchronization with source code in the specific language. The PIM profile contains 7 stereotypes with an average of 2 tagged values per stereotype. As an addition to that, both profiles contain data types, which are to be used as types for different attributes of classes in the models created with the profiles.

### 3.2 Operation

The study was performed in four stages. Each stage had a fixed design that was established based on the results of the previous stage, thus making the whole study a flexible design [14]. The study steps and their purpose are summarized in Table 1. Different objectives characterize different stages. Although there was a common goal – to cross-validate findings from other stages.
<table>
<thead>
<tr>
<th>Stage</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial interview</td>
<td>The main objective of the initial interview was to gather information on the purpose of the project, its scope and the necessary documents to investigate.</td>
</tr>
<tr>
<td>Document investigation</td>
<td>The documents were examined from the perspective of potential problems, inconsistencies and improvement issues. The goal was to elaborate an initial set of factors important in the profiles and transformations development in the MDA context.</td>
</tr>
<tr>
<td>Structured interview</td>
<td>The main objective was to identify more factors, especially important from the perspective of a developer of JNX and to establish a consensus on the found factors. It was also important to gather additional data important, but yet not present in the documents.</td>
</tr>
<tr>
<td>Observation</td>
<td>The purpose of the observation was to investigate whether the identified factors occur during the normal working conditions with the new framework and its development. The goal was also to minimize a possible subjective bias in the responses gathered during both interviews.</td>
</tr>
</tbody>
</table>

| Table 1. Stages of the case study |

The first stage was performed in the middle of the project lifecycle. That influenced the completeness of some of the artifacts. Any changes in them were verified during the third and the fourth stage. The investigation of the documents was done over a period of time and the findings were consulted with other documented experience reports on the industrial MDA applications. The documents (artifacts) were examined until the end of the project, where updated versions of the changed artefacts were obtained. The findings from the document investigations and the initial interview were used to elaborate questions for the structured interview. Most of the questions were multiple-choice, but the open questions were also used wherever possible. The last stage – the observation – was performed to minimize the bias of the respondents in the study and to identify other aspects of the usage and definition, which could not be uncovered during other phases. The whole study was performed in a three months time span.

### 3.3 Objects

The objects of the study were the set of artifacts of the JNX project. The artifacts included the following elements:

- UML Profiles that consist of models of profiles and documentations.
- Model transformations, presented as a compiled solution with source code and documentation.
- Tools used in the development of the solution and tools for which the solution was developed.
- Documentation on how to use the framework.
- Documentation on how to use additional tools for development of different JNX framework elements.

The objects were extensively used in the document investigation and observation phase. They were also a basis for certain questions in the structured interview part. Some of the objects were used during parts of the initial interview (some documents were provided at the beginning of the initial interview).

### 3.4 Validity evaluation

As all empirical studies, this case study has certain threats to its validity. The threats are presented in four groups according to [15]. Each group gathers threats that influence the same aspect of the study.

The external validity threats refer to the external generalizability of the study, describing the potential aspects of the specific study settings. The main threat in this
category is that the case study results might not be generalizable to other similar projects or companies because this was a specific case. To minimize such an effect, the results were cross-validated with the existing MDA experience papers (for example [12, 13, 16-18]). The identified factors were also checked whether they only occur in the case of this project. It was done, by the observation of the way of working of a user and a developer of the solution. Since the realization of the MDA approach was based on profiles, the results are generalizable for the same approaches. This threat concerns only some factors that are related to the profiles definition and usage (although not all). This threat essentially does not influence the generalizability of the transformation related factors.

One of the internal validity threats is that there are some aspects that were not investigated during the study. The overall flexible design with fixed stages allowed flexibility in adjusting to the current situation. The study on the literature to check for other factors was also performed to extend the set of possible factors, which were then again validated during the structured interview. Nevertheless, it seems that only some of the modeling aspects are investigated (due to the scope of the JNX project), namely the structural aspects. The behavioral aspects of software are captured in the JNX project in form of the black-box reusability and were investigated only partially, since it is not in the heart of model driven development.

An important aspect of the construct validity is that the design of the study might influence the results. In the case of this study, the threat was minimized by consulting other experience papers on MDA applications. It was conducted before the structured interview. The results were then validated during the interview to avoid finding results that are not relevant in projects similar to the JNX project. A lot of attention was paid to minimize threats of “fishing for results” – i.e. limiting the set of factors found to the factors already established in literature. Another threat to construct validity concerns the choice of the case to study. It was chosen based on the team’s extensive knowledge on MDA, the advanced realization of MDA, scale of the project (which can be considered as large and company-wide) and an extensive cooperation with the supporting UML tool vendors.

One threat to the validity of conclusions is related to the qualitative analysis of the obtained data. However, in the flexible study like the one performed, especially since it was an exploratory case study, qualitative analysis provides more flexibility in interpretation and allows consulting the results with the subjects that took part in the study. Furthermore, some quantitative methods were used during the document investigation to establish a baseline of the complexity and quality of the documents. Although meta-modeling literature was used in the process of cross-validation of the factors, it was used with great deal of cautious – especially from the perspective of differences between meta-modeling and profiling.

4 Factors influencing the effective realization of the MDA approach

The studied solution and the literature studies allow identification of a set of factors determining the effective realization of the MDA vision. The factors are grouped into two main groups, determining the effectiveness of:

- definition of the profiles and transformation in MDA, and
- usage of the profiles and transformations.

The factors gathered are complementary, although they influence two different stakeholders – creators of profiles and transformations and developers using the profiles and transformations.

4.1 Factors determining the effective definition

One of the perspectives, which were considered in course of the case study, was the perspective of the creators of the MDA process framework, including the associated methods and tools. The identification of the factors was done via documents
investigations and interviews with the staff members involved in the realization of the JNX framework. It was also supported by the current literature investigations in the area of MDA, meta-modeling and UML customization.

The study resulted in identification of the 11 factors affecting the definition and development of profiles and transformations.

Factor 1. **Clear definition of the purpose and scope of the realization of the MDA approach.** The clear definition of the purpose of developing the concrete realization should be established at an early stage of tailoring of the MDA principles for the endeavor. It should be precisely stated which methods, techniques and tools should be used in the course of the improved software development process – based on the MDA principles. The scope of the MDA realization should be defined in an unambiguous way – the activities in the software development process which are replaced by the transformations must be identified; different kinds of models that are used should be identified and UML profiles for them should be defined.

Factor 2. **Precise and unambiguous definition of profiles used.** It is important that the definition of the profiles for PIM and PSM models is precise [3]. The purposes of the customization should be investigated and the information that is needed for automatic transformations of profiled models should be elicited. The information should be either accessible as part of the model element that is transformed (for example the name of a model element) or it should be added as a tagged value of a defined stereotype (for example a kind of a persistency mechanism to be used). The stereotypes should contain constraints which prevent the illegal usage of the stereotyped elements. The illegal usage might be such usage in conjunction with other elements that it is not possible to automatically transform such constructs.

Factor 3. **Proper identification of the elements used in the PIM and PSM models.** The identified elements influence the definition of the profile in three aspects:

- some elements are used as part of the customization of the language,
- some elements are used without customization (non-stereotyped elements), and
- usage of some modeling constructs (a set of model elements) should be prohibited, since they do not belong to the specific domain of the PIM used.

The proper identification of elements determines whether information required in course of the automatic model transformations is accessible. This in turn influences the amount of additional properties of different model elements. If the elements are not properly identified, the transformations might be more complex.

Factor 4. **Presence of data types (or reusable classes) as an addition to stereotypes.** Since the stereotypes in the profile are domain (or solution) specific, the types of tagged values defined as part of the stereotypes should be alike. Therefore, a set of data types used in tag definitions should be part of the profile. Although the UML specification states that a profile should contain only stereotypes, there is no formal restriction on that as the profile is a special kind of a package (denoted as a specialization in the UML meta-model [2]).

Factor 5. **Concise profiles.** The purpose of the profiles is that they should provide only the necessary constructs to express the PIM or PSM models, they should contain:

- additional information for transformations (as tagged values),
• constraints on the usage of profiled models (as constraints), and
• a set of data types (or classes) that could be used by the profiled model elements.

The last bullet allows the profile to contain a set of reusable elements, which are to be used in virtually every model used in the profile. The elements could be extended in the specific models by the inheritance mechanisms.

Factor 6. **Automatic checking of the constraints on the profiles.** To be able to take the full advantage of the constraints – restrictions on model elements – they should be automatically checked during model development [19]. First of all, such constraints should be defined in a human-readable form and then they should be expressed in a formal language that can automatically be checked – for example in OCL [7].

Factor 7. **Black-box reusability.** Certain elements should not be allowed to be used explicitly by developers in the diagrams of UML models. Such elements should be enclosed in a reusable, shared library that is linked to the generated code. Models of shared library are linked to PSM, and some generated classes inherit from base classes to this model. In such case developers are prevented from modifying the functionality of the reused elements. An example of such situation is the usage of such mechanisms as logging and transactions, when the developers should only use the mechanism, while the redefinition of the element should be prohibited to them.

Factor 8. **Division of transformations into atomic or complex (such as whole structures of model elements – associations, aggregations between elements, etc).** Each well-formed\(^2\) model element from an input model should be possible to be transformed into a well-formed element in the result model. Each atomic transformation should transform one element into another element. A complex transformation should transform one element into another element and it should use atomic transformations to transform all elements that are part of the transformed element. An example is a transformation of a class, where the complex transformation for the class uses atomic transformations for attributes and operations to transform the class itself and all its parts. Such separation of transformations allows simplifying their definitions as also identified in [12].

Factor 9. **Precisely defined pre- and post- conditions for transformations.** Each transformation should precisely state the conditions on the input model element and the output model element. Such conditions might be complex, but it can be simplified with the help of stereotypes. If the input model element is stereotyped, the transformation should assume that the input model element is well-formed with respect to both the UML syntax and the stereotype constraints. The stereotype constraints should specify the static well-formedness rules on each stereotyped model elements, thus preventing the usage of such model elements improperly. If the elements are used properly, then is should be possible for the transformation to transform them.

Factor 10. **Traceability links added between the input and output elements of transformations.** The links allow tracing changes made in the designs during the development. What is more, the links provide information on the transformations that were performed. It is also easier to check which model elements were used to create a specific model element. When an error is encountered in the design, it is easier to trace the source of it. Together with the mechanisms of updating the

---
\(^2\) The well-formedness rules of the profile and the UML itself should be checked.
models via transformations (refinements), it provides a flexible technique of introducing changes in the designs [12].

Factor 11. Iterative and incremental small-scale and lightweight software process for profiles and transformations. The iterative and incremental development provides more room for early verification and validation of the solution, which is especially needed when the set of utility transformation is involved – the users can check whether the solution improves their work.

4.2 Factors determining the effective usage

The factors gathered in this group influence the way in which the solutions are used. They determine whether the introduction of the newly created way of software development is successful and used in the organization. The baseline which was used as the starting point for factors identification was the same software development environment, but without neither the automatic model transformations nor the specific UML profiles.

Factor 12. Integration into the currently used tool. The effectiveness of the solution is determined by a flat learning curve when introducing a new method or technique. In the case of using model transformations and profiles, it is important that they are properly introduced into the currently used tools. Despite a decreased portability of the tool dependent solution [16, 20, 21], certain benefits can be observed, for example, the transition to the new solution is easier – the users do not have to learn a new way of working with a new tool for model transformations [13, 22].

Factor 13. Improvement of the current way of working. It is one of the most important factors determining whether the new solution – the set of transformations, profiles and MDA-based technologies – is to be successfully introduced. If the introduced solution is making changes in a counter-intuitive and drastic way, the productivity is most likely to be lower. What is more, the users should be aware how the model transformations work, in order to be able to track possible errors encountered in late phases to the problems that lead to these errors (possibly in earlier phases).

Factor 14. Extensive documentation. It should be clear to the user how to use the new solution. The documentation should contain several elements, such as:

- documents presenting how to use the solution for the users that already work with the solved problems, but manually,
- documents on how to use the solution for the newly introduced users,
- troubleshooting documentation

The proper documentation is such projects as the internal process and method improvement significantly decreases the maintainability costs. Especially, the most profitable is to build the troubleshooting information database so that the recurring problems are solved only once.

Factor 15. Simple profile definition. It is important that the users of the solution understand all the information that needs to be added to stereotyped elements. If this is not the case, then some of the information in tagged values might be omitted – resulting in errors in transformations – or the constraints evaluation results might be hard to fix. Simpler profiles also ease grasping mentally the scope of the automatic help from the MDA framework [13]. Furthermore, a choice between using the profiles and meta-models is to be made. Profiles can improve understandability and introduce simplicity to the designs [3].
Factor 16. **Simple process framework definition.** The definition of the way in which the newly created MDA solution is to be used should be straightforward and it should be based on the context of the existing solutions. If it is described with references to the existing solutions, it is easier for the users to understand what kind of improvements the new solution brings. Especially for these users that are accustomed to working with the “old” solution. The description makes it easier for them to follow the automatic process with a higher awareness.

Factor 17. **Traceability of the transformations’ actions.** Information on the elements that were changed by the transformation should be provided. It should be clear how to trace the actions of the transformation back. Such information can be provided in a tool specific way as bindings, trace dependencies or similar (for example hypertext based links [3]).

4.3 **Summary**

The identified factors seem to be the most common and the most important determiners of a successful and efficient realization of Model Driven Architecture principles in an industrial setting. The factors determining the effective usage of the newly introduced are more significant in cases where the users of the solution are not the same as its creators. The factors affecting the definition and thus the implementation of the solution are particularly meaningful for the team that works with the development of MDA based frameworks.

5 **Related work**

Since the establishment of the model driven development according to the Model Driven Architecture principles, several studies were performed evaluating the newly introduced guidelines. Certain aspects related to improvement of productivity by the new platform were presented in [13]. The authors of that paper identified an improvement in the software delivery time and in software quality. It was obtained by an experiment in an industrial setting which compared the two methods via development of a small-size application with two independent teams using different approaches (standard and MDA-based).

A presentation of a similar case study can be found in [3], with the focus on the presentation of the MDA approach in the web application modeling. Despite a specific context of that study, some of the results were used in the process of cross-validation of the results. Nevertheless, the study described in that paper does not identify factors important for MDA. The study is also performed from a single – user – perspective.

De Miguel et al. [12] contributes with an interesting discussion on the decision between using MOF [23] based meta-models or UML profiles for the adoption of the MDA. The conclusion is that the UML profiles should be used whenever the extension of the language does not require a complete redefinition of certain elements. It is also complemented with the significantly larger support for the UML extension mechanisms than for meta-modeling. The paper also investigates important features of the UML tools which aid the creator of the MDA based frameworks.

Finally, some demands from tools to enable effective implementation of transformations are presented in [22]. The advantages and disadvantages of different approaches – ranging from the implementation directly in the UML tools, through a set of tool independent methods to the dedicated meta-model based transformation software – are presented in the paper. Nevertheless, the paper does not explicitly identify factors, mainly due to its general scope and summarizing character.

6 **Conclusions**

The paper presents an industrial case study on the realization of the Model Driven Architecture principles, techniques and methods in industry. The studied project was
successfully developed and is to be used as a basis for the subsequent MDA realizations. The successful development of the experimental project provides valuable guidelines and allows identifying factors determining the effective MDA realization in industrial context. During its lifecycle, a set of interesting problems, issues and challenges were encountered that are crucial for the success of the endeavor. In the case study, an examination of the project was performed from the perspective of possible factors that influence and determine the effective realization of such endeavor. The scope and the context of the project were advanced enough to provide a good baseline for the factors identification. The findings were cross-validated with literature survey on the existing experience papers on MDA realizations to ensure a more general scope of the findings.

The research performed within the study resulted in the identification of seventeen factors that affect the realization of MDA in industrial contexts. The factors were separated into two categories – usage and development of such endeavors. The two perspectives of examining the findings provided a possibility to identify the main challenges in the development and the usage of MDA. The main challenge of the definition of UML profiles and transformations in MDA is the proper abstraction and definition of the scope of the endeavor. The effective usage of the introduced MDA concepts, methods and principles is mainly determined by a proper and intuitive customization of the tools in the development as well as the simplicity of profiles and transformations. Due to the context of the study – industrial projects – certain aspects are not present. One of them is the discussion on the usage of meta-modeling for PIM and PSM modeling language creation, which was not included due to its lack of practical application caused by a marginal support in UML tools used in large industrial projects.

The further research directions are targeted towards identification a set of methods useful in tackling the problems appearing when the identified factors are taken into account. In the studied case, the solutions and improvements have a tendency of being too specific. A set of more general proposals is to be elaborated and evaluated through a series of experiments. Furthermore, other industrial MDA application cases are to be examined to gather more data points of the existing problems, issues and potential solutions to them.

References
The Coral Modelling Framework

Marcus Alanen and Ivan Porres
TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FI-N20520 Turku, Finland
e-mail: {marcus.alanen, ivan.porres}@abo.fi

Abstract The technology to fully support model-based software development approaches such as OMG’s model driven architecture is still in its infancy. There is still a lot to be learned about how a modelling framework should be constructed and used to enable using models as the only description of a software under construction and model transformation as the basic step in software development. In this short paper, we describe Coral, our own implementation of a modelling tool and some discoveries related to modelling and metamodelling that we have found.

Keywords: Modelling Frameworks, Model Driven Engineering, Metamodelling, Modelling

1 Introduction

The advance of modelling techniques both in academia and industry has lead to the development of several commercial modelling tools. In this paper, we present our work on Coral, a generic open source tool for modelling, and how we have been able to experiment with novel ideas in modelling. The research area of modelling tools is important as solid frameworks are required to empower software developers to actually use the benefits of a model driven architecture.

In the next section, we go through the more important features of Coral, and how it manages to create a flexible approach to querying and manipulating models. We finally conclude with some remarks on what features we consider important in a modelling framework.

2 Coral Features

Todays de facto modelling environment adhere to the specifications defined by the Object Management Group (OMG) more or less closely simply because of practicality; they have the broadest audience. OMG defines a modelling environment in four layers which are, from highest to lowest, the metametamodel, the metamodel, the model and the runtime layers. Each layer serves as a description of what can be accomplished in the layer immediately below it, akin to class definitions providing which objects can be created in an object-oriented language. The metametamodel is fixed to the Meta Object Facility (MOF) [6]. The most widely known metamodel is the Unified Modelling Language (UML) [3].

While UML has created useful ways to describe software systems in a visual language, the semantic preciseness of metamodelling and modelling constructs has been lacking. Unfortunately the various specifications do not help. Coral is an attempt to seek
practical and theoretical issues in metamodelling and modelling standards and outside standards and provide working solutions. In the next subsections various aspects of Coral are brought forward that show how it fits as a researcher’s tool.

2.1 A Dynamic Metamodelling and Modelling Tool

The Coral framework is based on few but important principles. The most fundamental is the notion of being metamodel-independent, i.e., Coral positions itself at the top of OMG’s layers creating a metametamodel interface. Using it, metamodels and models can be created at runtime. In several other modelling tools, there is only one or a few static metamodels to choose from; in Coral, metamodels are first-class citizens. Large parts of Coral try to be as ignorant of the underlying metamodel as possible, and several interesting algorithms and problems arise from this. The Coral metametamodelling layer is static, which has the implication that users cannot experiment with new metamodelling techniques. While the OMG standards are self-referencing and self-defining, this is usually not possible to do in a software program, so there is a limitation to the level of flexibility that can be constructed.

Even though Coral can create any metamodel at runtime, there are still some fixed metamodel elements (metaelements) for primitive datatypes such as integers, strings and floating-point values. To represent diagrams, the XMI-DI [7] metamodel is supported.

As the de facto serialisation format for models is the XML Metadata Interchange (XMI) [5], no metamodels (as defined in Coral) per se can be loaded or saved. Trivially this is rectified by noting that every metamodel can be interpreted as a model which therefore must have a metamodel. This metamodel is called the Simple Metamodel Description language (SMD) in Coral, and then metamodels can be represented as SMD models. SMD can be seen as analogous to MOF. The SMD metamodel is used to load models, from which metamodels can be created using a special routine, model2metamodel. This arrangement can be seen in Figure 1. Similarly, the metamodel can be transformed back to a model (which has SMD as its metamodel) using metamodel2model. This is portrayed in Figure 2.

![Figure 1](image1.png)

**Figure 1.** Lifting a model to the metamodel layer. The Simple Metamodel Description language (SMD) is statically linked into Coral. Using it, other metamodels can be loaded as models and then transformed into metamodels.

Naturally this arrangement creates a chicken-and-egg problem in practice with respect to the SMD language. This is circumvented by bootstrapping Coral with a hand-written SMD metamodel which is statically linked into Coral.

At the moment, there is no explicit support for the lowest layer in Coral. However, this is not a problem, since a user can transform (parts of) their model into an SMD model,
which can in turn be transformed into a metamodel $M$. Thus, objects in the runtime layer can be simulated by a model with a suitable metamodel. Having generic support for two layers (the model and the metamodel) seems to be enough, although we do not have the empirical evidence to support this argument yet.

An interesting feature of the dynamic nature of the metamodelling layer is the concept of importing the contents of one metamodel into the namespace of the current metamodel. This allows us to form hierarchies of metamodels. For example, a tool vendor uses its own namespace for the combination of UML 1.4 and XMI-DI 2.0. In Coral, this compatibility is achieved by creating the metamodels separately and then importing their contents into a third metamodel.

The modelling layer provides support for transactions by recording changes to models. Each transaction consists of an ordered list of commands. Trivially, a transaction can be undone and redone by traversing the list backwards unexecuting the commands or by traversing the list forwards executing the commands. Independent transaction observers can be attached (in a subject-observer design pattern). The graphical subsystem uses the transaction facility to automatically update the graphics when the model is changed by e.g. a script.

### 2.2 Mutually Independent Property Characteristics

In our opinion, the expressive power of metamodels does not come from the actual metaelements, but rather from the different characteristics of the interconnections between metaelements. An element’s possible connections (slots) are described by its metaelement’s properties. Two properties can be connected together to form a bidirectional meta-association.

In Coral, a property consists of several characteristics and describes the restrictions for each slot. Using a combination of characteristics several common constructs can be modelled, as well as more esoteric ones. It is important to notice that this part is static in Coral, i.e. users cannot change what characteristics are available, but are free to combine them in arbitrary ways. The various characteristics are described below.

- a *name* for convenience
- a *multiplicity* range $[l, u]$ defining how many connections to instances of the target the slot (instantiated property) should have to be well-formed. Common values are $[0..1]$ for an optional element, $[1..1]$ for exactly one, $[0..*]$ for any amount and $[1..*]$ for at least one element
- a *target*, telling what the type (metaelement) of every element in the slot must be
- a boolean *ordered* telling if the order of the elements in the slots is important and must be kept

![Figure 2. Lowering a metamodel to the model layer, the opposite operation of Figure 1.](image-url)
- a boolean flag \textit{bag} telling if the same element can occur several times in a slot
- a boolean flag \textit{anonymous} telling if the property is anonymous (described later)
- a boolean flag \textit{unserialisable} telling if corresponding slots should not be serialised when saving a model
- an optional \textit{opposite}, giving the opposite property for bidirectional connections
- an \textit{link type} enumeration value \{association, composition\} describing an ordinary connection or describing ownership, respectively.

The characteristics \textit{unserialisable} and \textit{anonymous} need more careful explanation. An unserialisable property means that the contents of the corresponding slots are not saved to an output stream. This is useful when elements in file A reference elements in file B, but without the elements in B having to know anything about file A. This occurs when creating models that resemble “plugins”; we are not sure what plugins are available and we do not want to change the main file every time something is added or removed. Instead, the available plugins are loaded at runtime and bidirectional connections are created, even though they are not serialised at both ends. Arguably the usefulness of the characteristic in this case is specific to the way current filesystems work using files as independent streams of bytes. A filesystem acting more like a database would not share the benefits from the unserialisable characteristic.

Anonymous properties provide fully bidirectional meta-associations between any metaelements even though the meta-association was unidirectional at first. This is useful in cases where a metamodel was not designed to be used together with another metamodel. An example is a project management metamodel (PMM) keeping track of developers, bugs, timelines and several UML models. Since UML models do not know about PMM, only unidirectional connections from PMM to UML would be feasible, thus rendering any navigation from UML models to PMM models impossible. But Coral automatically creates an anonymous property (with a private, nonconflicting name) at runtime from UML models to PMM models, and thus it is indeed possible to navigate from any UML model to the corresponding PMM model(s). The UML models can then be saved in one file, and the PMM models in another; the XMI standard for model interchange contains facilities for linking across files. Anonymous properties are necessarily also unserialised, since otherwise ordinary UML tools would not be able to read the UML model file with nonstandard slots.

Most notably, the list is currently missing new characteristics from MOF 2.0, property \textit{subsetting} and \textit{derived unions}. These are important characteristics but have not been added to Coral yet. Otherwise, it is worth noting that the characteristics aim to be as mutually independent as possible. This has the benefit that very complex definitions can be modelled.

\subsection*{2.3 Python Scripting Interface}

An important feature of a modelling framework is its ability to query and modify models at runtime, preferably both interactively and using a script. In Coral, this has been achieved by creating Python wrappers around the Coral C++ core. Python is a highly dynamic expressive language which is easy to learn. Using Python, the interface to query models is very close to OCL [2], but with several methods added to also modify the model.

Notably, model transformations can be written as Python programs with separate phases for preconditions, query and modification and postconditions. Support for transactions as well as checking of well-formed rules means that an illegal transformation can
be rolled back, leaving the user with the original model. Examples of a rule-based model transformer can be found in [9].

Arbitrary scripts and well-formedness checks can be used to keep the design and evolution of a system within a predefined process or methodology. A success story is Dragos Truscan’s work [10] on relations between data flow diagrams and object diagrams. It presents “an approach to combine both data-flow and object-oriented computing paradigms to model embedded systems.” The work is fundamental for designing complex embedded systems since there is often a need to switch between the two paradigms. The design relies on an SA/RT metamodel for the data flow and the UML 1.4 metamodel for object and class diagrams. Python scripts are heavily used for the transformations between the domains.

In the future, using models also as the primary artefact for transformations using e.g. the upcoming OMG Query-View-Transform (QVT) [4] standard could be possible.

The scripting interface provides a highly flexible environment for automatic model generation, querying and transformation. SMD metamodels support pre-defined operations on specific elements, and there is no need to explicitly compile any scripts as they can be loaded on-the-fly from within Coral.

2.4 Miscellaneous

Coral supports XMI 1.x and XMI 2.0 input and XMI 1.2 and XMI 2.0 output well. It also has support for more esoteric features such as interfile relationships although these are not too heavily tested. In practice it has good support for reading XMI generated by other common commercial tools.

Coral currently comes with the UML 1.1, UML 1.3, UML 1.4 and UML 1.5 metamodels, but interactive graphical support is lacking. How the presentation (graphics) of a metamodel is defined and drawn has not been as thoroughly developed by OMG as the abstract metamodels, which unfortunately is reflected in Coral as well; support for every diagram must be written explicitly, although there is work-in-progress to use models to generate graphical interfaces.

Coral has support for difference calculation between two models, based on [1], but this has not been integrated into the graphical environment.

Currently, Coral and its predecessor SMW [8] have been used as modelling and metamodeling tools by two PhD students and several Master’s Theses have been written on their plugins and subsystems.

3 Conclusions

We have presented the modelling tool Coral, its main features and primary principles, as well as some novel concepts that it uses to address practical problems in the creation of such a tool. Coral is still work-in-progress, but is used by other members of our model driven engineering group more and more.

Coral works as a metamodel-independent tool. For this to be possible, metamodels must be treated as first-class citizens that can be created at runtime. Furthermore, we have made interesting progress on the characteristics of metaelements’ properties in the form of anonymous properties, which greatly aid linking together metamodels. Effort has been placed into making a Python-friendly interface to facilitate easy scripting. However, there is a lot of on-going work with e.g. the interactive part of Coral.
References

Putting Concern-Oriented Modeling into Practice

Stéphane Bonnet\textsuperscript{1, 2}, Raphaël Marvie\textsuperscript{2}, Jean-Marc Geib\textsuperscript{2}

\textsuperscript{1} Gemplus, Systems Research Labs, La Vigie, ZI Athelia, La Ciotat, France
steephane.bonnet@research.gemplus.com
\textsuperscript{2} Laboratoire d’Informatique Fondamentale de Lille, UMR CNRS 8022
Université des Sciences et Techniques de Lille, Villeneuve d’Ascq, France
\{raphael.marvie,jean-marc.geib\}@lifr.fr

Abstract. Providing a methodological support for designing software, separation of concerns particularly suits the emerging model paradigm. This paper elaborates on concern-oriented modeling, an approach in which the organization of models and meta-models is based on preoccupations. After a description of an annotation-based framework for modeling concerns, this paper focuses on related model transformation issues. The last part of the paper introduces and illustrates a design pattern for model merging in the context of concern-oriented modeling.

1 Introduction

The work presented in this paper is motivated by smart card configuration issues. The very personal and secure nature of smart cards makes their configuration a complex process. The customization grain ranges from huge clients such as telecommunication operators to individual card holders. While the forthcoming evolutions of smart card systems and applications are calling for advanced customization solutions, we are experimenting new configuration methods [1].

The last evolutions of software engineering are characterized by the success of model-driven approaches. Designing a system is progressively becoming a matter of model specification, organization, and manipulation. However, model-driven engineering is still in its early stages and is therefore raising very challenging issues. Among them, structuring efficiently the models and implementing powerful transformations are two key success factors.

At the core of software design for many years already, the separation of concerns principle allows to reduce the complexity of building software systems [2]. This principle is driving the design of our smart card configuration architecture, as explained in [1]. This paper sketches out our understanding of concern-oriented modeling, an approach that consists in decomposing the design of a system in distinct concern-specific models. Throughout the paper, we illustrate our reflection with an example taken from our smart card configuration architecture.

Sect. 2 presents the current model engineering landscape, introduces the separation of concerns concept and describes a framework for designing concern-specific meta-models. Sect. 3 elaborates on related model transformation issues, and describes two approaches particularly interesting for concern-oriented modeling: marking-based trans-
formation and model merging. The last section describes and illustrates a proposal for handling model merging.

2 Separation of Concerns and Meta-Modeling

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) raises the level of abstraction of software development by making models first-class entities of the design process [3]. Because their interpretation must be unambiguous, models are associated with a unique semantics defined by their meta-model. Meta-modeling allows to define the concepts that can be used to model systems: a meta-model is a language to write models, and at the upper layer, a meta-meta-model is a language to write meta-models [4].

Multiple meta-models may be created for various purposes. The structuring of their definition and uses remains an important issue of model engineering. In 2001, the Object Management Group (OMG) has introduced MDA, a specification standardizing model-driven software development [5]. The core foundation of MDA is the distinction between functional and technological preoccupations: while platform-independent models (PIM) describe system functionality, platform-specific models (PSM) express the technological specificities of its implementation.

However, adopting a model-driven approach to design a software system requires the capture of different views of this system, at variable levels of abstraction, with different preoccupations in mind. Despite being a major issue to consider, the dependence on a platform is therefore only one preoccupation among many others. It is essential to widen the model classification criteria proposed by MDA [6] [7].

2.2 Separation of Concerns

Concerns are among the primary motivations for organizing and decomposing software into manageable and comprehensible parts [2]. Reasoning with concerns allows to tackle the specification of a system as a composition of multiple views corresponding to multiple actors with different preoccupations.

Separation of concerns is actually a ubiquitous principle of software engineering that has led to the invention of many modularization approaches. However, the understanding of the notion of concern is still not well-established, in particular because the separation might occur in various contexts:

- Between functional and technological concerns. This kind of separation enables a clear distinction between the design of the functionality of a system, and its projections on different executing environments. As described in Sect. 2.1, MDA relies on this kind of separation.
- Between different functional concerns. A system might be functionally decomposed in several subsystems that address subset of features.
- Between functional and non-functional concerns. Such a separation allows a transversal handling of non-functional issues such as security, persistence, etc. Crosscutting concerns are called aspects, and form the basis of aspect-oriented programming [8].
- Between different phases of the engineering process. The decomposition of a software production process in several engineering activities is a kind of separation of concerns.

A global definition of the separation of concerns principle is provided by [2]: "In its most general form, separation of concerns refers to the ability to identify, encapsulate, and manipulate only those part of software that are relevant to a particular concept, goal, or purpose". Such a decomposition strategy provides a pertinent methodological support
to classify models and meta-models, and thus structure model-driven approaches [7] [9].

The reminder of this paper focuses on a possible implementation of this principle: after
the presentation a framework for modeling concerns, we elaborate on some related
model transformation issues.

2.2 A Framework for Concern-Oriented Meta-Modeling

The framework described here is a subset of CODeX, an approach for structuring the
definition and the use of MOF meta-models [9]. CODeX proposes an organization of
meta-models based on three levels: the first one represents the definition of the common
vocabulary of a domain, the second one contains one annotation plan per concern of the
system, and the third one realizes the integration of the different concerns. Each annota-
tion plan contains two meta-models (“M2” stands for Meta-Model):
– The meta-model M2_Base that defines the core concepts of the concern,
– The meta-model M2_Annotations that defines how the concepts of M2_Base
relate to the common vocabulary. M2_Annotations is roughly "the sum" of
M2_Base and M2_Domain meta-models.

Fig. 1 illustrates the two first levels of the framework, with two annotation plans repre-
senting two concerns called C1 and C2. It also shows how a given M2_Annotations
package both imports the M2_Base package and inherits the M2_Domain package.
Because all meta-models are MOF packages, CODeX relies on the package inheritance
concept defined by the MOF [4].

![Fig. 1. The concept of annotation plans](image)

In a third level, CODeX specifies how the different annotation plans should be reunified
to provide a full representation of the system, but as our approach is slightly different,
we leave aside here the integration issues.

2.3 Example

The example introduced in this section will be developed throughout the reminder of the
paper. It is a personalization framework taken from the model-driven approach for smart
card configuration described in [1]. The goal of this framework is to automate the suc-
cessive model creations and transformations required to instantiate user-specific con-
figurations of given on-card applications. The experimental context of this research
explains that the meta-models are kept voluntarily simple.

To the M2_Domain meta-model of the previous section corresponds M2_Appli, a
basic meta-model for specifying applications. This meta-model can be seen as a very
minimal subset of UML and is only defined for experimental purposes. Fig. 2 only shows the concepts that are relevant to the annotation context. M2PersoBase and M2PersoAnnot meta-models describe a very basic annotation plan for the personalization concern. It simply introduces the concept of class, attribute and association personalizers. The way these meta-models are used is illustrated in section 4.3.

Fig. 2. The personalization annotation plan

The framework for concern-oriented modeling presented here provides means to decompose rigorously the specification of a system though different preoccupations. The real goal of such a framework is however to automate the processing of the different concern descriptions, as explained in the next sections.

3. Model Transformation Schemes

3.1 Model Transformations

MDE is meant to increase both productivity and quality of software engineering. The success of this vision is conditioned by the development of effective means for defining model transformations [10]. A transformation between two models roughly consists in sets of rules, which establish correspondences between elements of the source model and elements of the target model. There is a lot of thinking among the research community about the different approaches for transforming models [11] [12]. Considering transformations as models themselves is one of them.
In 2002, the OMG has posted a Request For Proposal (RFP) called MOF 2.0 Query/Views/Transformations [13]. This RFP aims at providing a unified transformation language (meta-model), standardizing the mappings between models whose metamodels are defined using the MOF. QVT also addresses the way of querying MOF models, and the way of creating view onto the models. The two final submissions are described by [14] and [15], the result is expected in 2004.

The OMG introduces several configurations of transformations for the PIM to PSM projections [16]. Leaving aside the PIM and PSM concepts, we elaborate here on marking-based transformation and model merging, two configurations we find useful in the context of concern-oriented modeling.

### 3.2 Transformations for Concern-Oriented Modeling

Marking-based transformations rely on the addition of marks to the source model elements. The semantics of the marks is not defined by the source meta-model, but in a dedicated meta-model. The transformation between the source and target models is guided by the added marks. This configuration of transformation suits the concept of annotation plan introduced in the previous section. Each concern is associated with a set of marks.

![Marking-based transformation](image1)

**Fig. 3.** Marking-based transformation

Given the notations of Sect. 2.2:

- The meta-model of Source Model corresponds to M2_Domain meta-model,
- The semantics of the marks is defined by the M2_Base meta-model of the concern annotation plan,
- The meta-model of the Marked Source Model is the M2_Annotations metamodel of the concern annotation plan.

Model merging is one kind of Y transformation. It consists in a rule-driven composition of elements from two source models. This configuration might be a projection or a refinement. In the latter case, the target meta-model is, roughly speaking, the sum of the two source models.

![Model merging](image2)

**Fig. 4.** Model merging

In the context of concern-oriented modeling, the Source Model and Another Source Model can express for example two different concerns of a same system. The merging configuration can be a solution for the problem of integrating the different models created for different concerns (weaving of concerns).
4 A Design Pattern for the Model Merging Problem

4.1 The Model Merging Problem

Model-driven approaches often require sooner or later in the engineering process to combine different sources of information. This configuration is often referred to as the “Y” cycle, where the two upper branches of the "Y" represent two input sets of information that have to be combined to result in one single set. For example, the global structure of the MDA is actually a "Y" scheme where:

− the left branch of the "Y" contains successive refinements of PIMs,
− the right branch contains Platform Description Models (PDM),
− the bottom branch contains the successive PSMs refinements, and the final code.

The binding step, which corresponds to the center of the Y, remains one of the most difficult and unanswered question about the MDA [17]. In the following, we concentrate on a concrete example of Y scheme, the model merging configuration introduced in section 3.2 (Fig. 4).

Most of the current research on model transformation focuses on one-to-one configurations. Despite they are using different formalisms, tools or practices, most approaches for handling one-to-one transformations rely on a common strategy: the elements of the source model are parsed and specific rules mapping rules are applied on them. In the context of model merging, it is impossible to parse simultaneously both input models. The two source models can therefore not be considered as absolutely equal regarding the transformation realization.

Our proposition consists in refining the Y scheme to obtain a more known configuration, depicted by Fig. 5. As the transformation base parsing can only be realized on one of the two source models, we propose to create a third model \( R(FirstSource, SecondSource) \) that establishes relations between elements of models FirstSource and SecondSource. The Y scheme then becomes a one-to-one parameterized transformation, where the actual source is FirstSource.
Fig. 5. Refining the Y scheme

4.2 Implementing the One-to-one Parameterized Transformation

An overview of our parameterized transformation meta-model is given by Fig. 6.

Fig. 6. Simplified meta-model for one-to-one parameterized transformation
As explained previously, a transformation can globally be considered as a set of rules that are defined for dedicated concepts of the source meta-model. The rules can be applied in a specific order, specified by a strategy. The result of the application of a rule is the creation of one or more elements in the target model. The most interesting parts of the meta-model are obviously the concepts of ParamLink and ParameterizationModel:

- A ParamLink object creates a link between an element of the first source model and an element of the second source model.
- The ParameterizationModel is a model that gathers all the ParamLink for two given input models. Each time a Rule is applied on a FirstSourceElement, the ParameterizationModel is asked for the possible existence of a ParamLink for this element. In case one is found, the rule logic realizes the treatment allowing the creation of one or several TargetElement.

The creation of the parameterization model is the trickiest step. It is heavily dependent on the context, and in certain cases, links between meta-elements may even be sufficient. It is therefore difficult to specify a pertinent meta-model for this parameterization model. Sect. 4.3.4 provides the detailed example of a possible approach.

As the QVT RFP is not ready yet, there is no standard for specifying and implementing transformations. We have therefore coded a set of Java classes that provide the core transformation mechanisms, in order to reduce as much as possible the required effort to design new transformations. Our pragmatic approach is based on a combination of transformation modeling (specification of the inputs, targets, rules), code generation (skeleton of the Java classes for the transformation), and handwritten code (logic of the rules).

The implementation of this ad-hoc framework is realized on top of the Java Metadata Interface specification [18], a framework to manipulate models. However, our goal is not to provide another generic transformation meta-model. We therefore consider that providing here further implementation details would be out of the scope.

4.3 Example

The objective here is to illustrate our approach to model merging with the personalization of a smart card application:

- The first model consists in the basic model of the application, enriched with personalization marks that were introduced in section 2.3.
- The second source model is a model describing relevant information about the final user of the application, i.e., the smart card holder.
- The result of the merge of these two models is a model of a personalized application activator, from which can be generated the bootstrap code of final the application [1].

4.3.1 First Source Model
We take the example of an application called LoyaltyManager, illustrated by Fig. 7. This small application for managing electronic purses and loyalty accounts is embedded on a smart card.

![Diagram of LoyaltyManager application model]

**Fig. 7.** Annotated LoyaltyManager application model

The following piece of XML code is an extract of the serialized version of this model, to which personalization annotations have been added. Instead of showing the whole annotated model, we are focusing on the personalization of the LoyaltyAccount class with M2ClassPersonalizer and M2AttributePersonalizer elements.

```xml
<M2Class xmi.id="10" name="LoyaltyAccount">
  <M2Class.attributes>
    <M2Attribute xmi.idref="15"/>
    <M2Attribute xmi.idref="16"/>
    <M2Attribute xmi.idref="17"/>
  </M2Class.attributes>
  <M2ClassPersonalizer xmi.idref="20"/>  
</M2Class>

<M2Attribute xmi.id="15" name="ID">
  <M2Class.marker>
    <M2AttributePersonalizer xmi.idref="25"/>
  </M2Class.marker>
</M2Attribute>

<M2Attribute xmi.id="16" name="Merchant">
  <M2Class.marker>
  </M2Class.marker>
</M2Attribute>
```
4.3.2 Second Source Model

This model gathers application-relevant information about the final end user, such as demographics, and preferences. The meta-model of this second source model is provided by Fig. 8.

The following piece of XML shows the elements that are relative to the personalization of the LoyaltyAccount class and its attributes. It states that the user Carla Diaz owns two purses and is registered in two shops. It also precise which purse is associated to which shop.
4.3.3 Parameterization Model

We have chosen to represent this model at the code level only, and to rely on the Tag properties defined in the elements of M2PersoBase and M2User meta-models. We have defined a Java class ParamHandler that is encapsulating the creation of the links between source models. The ParamLink elements introduced by Fig. 6 correspond in this context to the couples stored in the HashTable object myParamLinks. The createLinks() method parses both source models, and creates links between elements every time the Tag properties of two model elements is matching. The following piece of code details the implementation of the ParamHandler class.
import javax.jmi.reflect.*/
import java.util.*;

public class ParamHandler {
    private RefPackage myFirstSource;
    private RefPackage mySecondSource;
    // couples [first source elt -> second source elt]
    private Hashtable myParamLinks = new Hashtable();

    public ParamHandler(RefPackage fs, RefPackage ss) {
        myFirstSource = fs;
        mySecondSource = ss;
    }

    public void createLinks() {
        Iterator it1 = myFirstSource.refAllClasses().iterator();
        while (it1.hasNext()) {
            RefClass rc = (RefClass) it1.next();
            Iterator it2 = rc.refAllOfClass().iterator();
            while (it2.hasNext()) {
                RefObject fs = (RefObject) it2.next();
                String tagvalue = (fs.refGetValue("Tag")).toString();
                if (!(tagvalue.equals(""))) {
                    RefObject ss = extractParamElement("Tag");
                    if (ss != null)
                        myParamLinks.put(fs, ss);
                }
            }
        }
    }

    public RefObject extractParamElement(String tagvalue) {
        Iterator it1 = mySecondSource.refAllClasses().iterator();
        while (it1.hasNext()) {
            RefClass rc = (RefClass) it1.next();
            Iterator it2 = rc.refAllOfClass().iterator();
            while (it2.hasNext()) {
                RefObject ss = (RefObject) it2.next();
                try {
                    String tagval = (o.refGetValue("Tag")).toString();
                    if (tagval.equals(tagvalue)) {
                        return ss;
                    }
                } catch (Exception e) {
                }
            }
        }
        return null;
    }
}

Running this code with the two source models detailed previously results for example in the creation of links between the M2ClassPersonalizer "LoyaltyAccount" element and the M2ItemValue "Compustore" and "FashionClothes" elements. The rest of source models and thus the other parameterization links are not described here, but it also contains information about the instantiation of the Purse class, and about the correlations between Purse and LoyaltyAccount instances.

This ParamHandler Java object representing the parameterization model is created in the beginning of the transformation. When a rule is applied on an element, this object is asked whether there exist a parameterization link for the concerned element.
4.3.4 The Activator Model

The result of the one-to-one parameterization transformation is a user-specific configuration of the application. The model (Fig. 9) describes a personalized activator, which is used for generating the actual bootstrap code of the application.

![Model of a user-specific activator model](image)

5 Conclusion

Model-driven engineering (MDE) offers a migration path from present object- and component-based solutions to a more ambitious and scalable approach for designing software [3]. MDE is making many promises today, but important issues remain to be tackled. Among them, we are particularly concerned by the structuring and transformation problems.

Separation of concerns is among the major design considerations in software engineering. We have recalled the principle and shown how it helps with reducing the complexity of building software systems. The separation of concerns in its most general form provides a pertinent methodological support for classifying models and meta-models. It can enable an extension of model classification beyond the only platform dependence criterion proposed by MDA.

We have described a mechanism to structure the specification of systems through concern-dedicated annotation plans [7]. The approach described here shows that the CODEX concepts first introduced in [9] and illustrated with component-based architectures remain appropriate in a different context, i.e., with marking-based transformations as a weaving strategy. We finally proposed a design pattern to handle model...
merging as parameterized one-to-one transformation. Modeling the relationships between the two source models is a step which automation is difficult to achieve. As illustrated by the example and the correspondences between some elements properties, the existence of correlations between the two models can help.

The next step of this work is to evaluate how the approach can scale up when taking into consideration multiple concerns, and in particular crosscutting ones. The design of dedicated tools supporting concern-oriented modeling is also to be considered if the interest for the approach is being confirmed.

When prototyping, the main technical issues we had to tackle were relative to the implementation of model transformations. The result of the QVT RFP will hopefully come in a (relatively) close future. It will then be interesting to see up to what point it will change the current modeling landscape, and how the different approaches, including ours, will be leveraged or called into question.

References

1  S. Bonnet, O. Potonniée, J.-M. Geib, R. Marvie, A Model-Driven Approach for Smart Card Configuration, In the 3rd International conference on Generative Programming and Component Engineering (GPCE ’04), to appear
3  J. Bézivin, MDA : From Hype to Hope, and Reality, <<UML>> 2003, San Francisco, Guest talk, 2003
5  MDA, Model Driven Architecture. OMG Specification (2001)
14 QVT Partners, MOF Query/Views/Transformations, Revised Submission Available at http://www.qvtp.org/
JMI, Java Metadata Interface, Sun Microsystems Available at http://java.sun.com/products/jmi
Abstract. This article describes the architecture of a tool to generate complete three-tier web applications from relational databases. The tool includes a set of metamodels to support the whole process. Accessing the database dictionary, the tool creates an instance of the database metamodel applying a reverse engineering technique; then, via a restructuration process, it gets an instance of a class metamodel that transforms the relational schema into a class diagram; finally, and according to the user preference, through a forward engineering process, the tool builds a new instance of the class metamodel for representing the final application. A set of functions are defined at a conceptual level to make these transformations, which are implemented in the tool by means of a set of factories. The proposed metamodels are the core of our tool.

1 Introduction

According to [1], reengineering is the process of applying, in the first time, reverse engineering to an existing software system to obtain higher level structures representing it and, in the second time, a later stage of forward engineering to get a new version of the original system, probably with new functionalities and higher quality. So, from the first stage, a set of abstract specifications from the original system is got, that are used to build a new implementation of the software system (Fig. 1).

Although it is the most frequent, source code is not always the starting point of reengineering: in fact, several authors have studied and proposed techniques for reverse-engineering other types of products, such as databases. Their goal is to obtain a conceptual model representing the original problem domain as an entity-relationship [2-4] or a class diagram [5]. The use of class diagrams instead of entity-relationships schemas supposes, from the reengineering point of view, the possibility of taking advantage of the characteristics of the object-oriented paradigm, not only for the later reconstruction of the database, but also for the implementation of applications capable of manipulating the database.
This article describes the metamodels included in a tool to generate complete web applications from relational databases, and shows how they are integrated into the tool architecture. All relational databases share the same set of core elements, such as tables, primary keys, foreign keys or triggers; so, it is possible to represent them using a vendor-independent metamodel. However, each vendor provides a different set of data types, and also the way to recover the database structure can be different for each manager. In order to add the tool the capability of instantiating the database metamodel from different vendors, an abstract factory was defined with so many concrete implementations as different types of databases to deal. Once we have the instance of the database metamodel, a function is applied to get a new metamodel, now in the form of a class diagram representing the business tier of the web application to be generated. This latest class metamodel is again transformed into a new metamodel that depends on the target platform and that also represents the business tier of the final application. From this one, the presentation tier and the persistence tier of the target application is built and used to make the code generation. In our case, we can generate applications based on EJB components or on standard Java classes.

The article is organized as follows: Section 2 contains an overview of some reverse engineer techniques related with our study; Section 3 is the core of the paper and describes the metamodels and architecture of the generation tool; finally, Section 4 draws our conclusions and future lines of work.

## 2 Background

Most works on database reverse engineering are focused on recovering the conceptual schema used during its initial development to migrate the database from an old data model (Codasyl, for example) to a new one, to the change of database management system, to the detection of integrity errors, etc. In our case the goal is to get a web application to be able to manage its data.

Several authors have made different proposals for reverse engineering relational databases. Hainaut et al. [4] propose a general method to apply reverse engineering to any database system or files collection, consisting of four big processes, quite inverse to the forward-engineering stage: (1) Data structure extraction; (2) Data structure conceptualization; (3) Untranslational, that detects compliant constructs (with respect to the database management system) and replaces them by independent constructs; and (4)
Conceptual normalization, that recovers the high level structures produced during the forward stage.

Pedro de Jesus and Sousa [6] also presented an interesting study analyzing the characteristics of several proposals (required initial state of the database, output products...) and presented a method allowing the mixed use of all of them. Eight methods of different authors are analyzed: except one of them [7], that produces an OMT class diagram, the others produce ER or extended-ER conceptual schemas. These methods produce their best results when the legacy data fulfill a set of desirable preconditions, as to be at least in the 3rd Normal Form, lack of inconsistencies, etc. The method by Pedro de Jesus and Sousa takes advantage of all the previous ones because it obtains a set of database clusters, every one of them grouping elements according to their suitability for being reverse engineered with the use of one of the methods they analyze. At the end, their “macromethod” also produces an entity-relationship schema, that is used as starting point for the construction of the new database.

Andersson [2] extracts the possible entity relationship schema used to build the database analyzing the Data Manipulation languages, looking for even among the SQL statements embedded in the source code of the programs that use the database. So, for example, primary and foreign keys are identified through the study of the join conditions existing in the SQL statements.

As users generally manage the information kept in a database through a set of external programs, the structure of the documents where the database is represented (i.e., data models), has a strong influence on the corresponding documents representing the programs (i.e., functional models). The object-oriented paradigm increases the cohesion of both models unifying data and behavior under a common point of view. So, the entity-relationship schema obtained from the reverse-engineering stage could be translated into a class diagram in order to take advantage of object-oriented constructions, tools, etc. for the next forward stage. In [5], we described an algorithm to get a class diagram from a relational database; now, we have improved the metamodel and its integration into a tool to generate applications.

3 Structure and behavior of the tool

This section describes the structure of the tool, explaining how it works in each stage of the reengineering process.

3.1 Reverse engineering and restructuration

Fig. 2 shows the main window of our tool. As it can work with four different database managers (Oracle, Caché Intersystems, SQL Server and Microsoft Access), it has four different sets of widgets to connect them. When any of the “Reverse” buttons is pressed, the tool tries to connect the desired database and, if the connection is possible, it instantiates a suitable factory that reads its data dictionary and builds an instance of the relational database metamodel.
These factories are specializations of an abstract factory, that includes the `getBD` operation, which must be implemented in the different concrete factories. Although the tool is implemented in Java and makes use of the JDBC API (what includes a set of interfaces to provide an uniform way for managing databases), not all vendors implement all the operations in these interfaces. So, the code of each factory has small differences with the others.

Fig. 3 Set of factories to deal with the different databases

Fig. 4 describes the structure of the metamodel used to represent relational databases. In its current version, the tool considers the presence of tables, integral reference relationships among tables, columns and stored procedures. Each column keeps information about its name, whether is or is not part of the primary key and whether is or is not part of some unique index. It also keeps an integer value representing its data type (`mType` field), that depends on the vendor: Oracle and SQL Server, for example, assign different numeric values to the `VARCHAR` type. Later, in the stage of code generation, some type of mapping must be implemented to maintain the correspondence between the vendor’s data types and the data types of the final application.
In order to prepare the code generation process, the instance of the database metamodel is translated into an instance of a metamodel representing class diagrams (Fig. 5): the OOS class represents the Object-Oriented System that is created from the database reverse-engineered. As it is seen, each OOS knows its set of classes and relationships; each relationship knows two classes (the left and the right one, identified in the figure by the role names mLeft and mRight) and keeps information about the cardinality of each side of the relationship (fields mCardL and mCardR) and about the navigability (boolean fields mFinalLeft and mFinalRight). The last field in the Relationship class (mNumber) represents an unique number given to this relationship, that can be useful when the same class is related to another class through more than one relationship.

The Class class in Fig. 5 represents a qualifier of the class model and keeps information about its nature (interface or class) and about its abstraction. It also knows about the other classes in the class diagram that are its superclasses (mSuperclasses role), its set of fields, constructors and methods.

The tool builds an OOS from one instance of Database. Initially, it adds a class for each table and, for each column, one field is added to its corresponding class. Foreign keys among tables are translated into associations among classes. Those foreign key relationships in the database whose multiplicity in both sides is one are considered to be inheritance relationships; this may imply to remove from the base class those fields that are defined in the superclass. At this step of the reengineering process, the tool does not add any methods to the class. This is made in the next step.
An additional class is added to the system to allow the execution of the stored procedures defined in the database. A public method is added to this class for each stored procedure. The parameter types must also fulfill the set of data types of the programming language of the final application.

### 3.2 Forward engineering and code generation

The current version of the tool is capable of generating three-tier web applications: the presentation tier consists of a set of JSP and HTM pages that allow the interaction of the user with the data saved in the database; all operations executed by the user are sent to the corresponding class in the domain tier, that has an implementation of these CRUD methods [8]:

a) An “empty” constructor, that assigns each field in the class a default value, such as 0 for integers or null for strings.

b) A materializer constructor, that builds an instance of the class from a record in the database.

c) The insert, delete and update methods, that add new records to the database, delete them or update them.

d) “Setters” and “getters” methods are also added to each class, as well as other operations that allow the navigation among related records.

The persistence tier has just one class, a database Broker [9], that acts as a mediator between the domain tier and the database.

Thereinafter, before the code generation, the instance of the class diagram metamodel is translated into another instance of a platform-dependent class metamodel. In the domain tier of the final application, the tool can generate either Enterprise JavaBeans (EJB) components or standard Java classes. An EJB component is a special type of class: it has no constructors, may have an associated “primary key class” and is accessible to the client application via two interfaces that are managed by a container.

When a client desires to execute any kind of operation on an EJB instance, it uses the so-called Home interface of the EJB, that is exposed by the container; the container locates
the component and returns the client a reference to the so-called Remote interface; then, the client uses this one to call the business operations.

So, when the final application will be based on EJBs, the class diagram obtained from the database is translated into a new representation: a metamodel that explicitly considers Java structures, such as packages, import statements, throwing clauses, static fields and methods, and Java types. This one constitutes a specialized metamodel, whose structure is shown in Fig. 6.

Therefore, from each class proceeding from a table (i.e., a class of the metamodel shown in Fig. 5) the tool generates four instances of the JavaClass shown in Fig. 6 the EJB class, the primary key class and the two interfaces (remote and home).

3.3 Example

Let us suppose a database with two tables, Person and Car, that are related via a foreign key relationship to know the owners of every car. The database schema could be this one:

![Fig. 7. A simple database](image)

After the tool has reverse-engineered the database, it builds one instance of the class metamodel. Fig. 8 shows a snapshot of the Oracle JDeveloper debugger after having reverse-engineered the database: it has a Vector field with two instances of Class inside and one Relationship object. Each class preserves the table name and has so many fields as columns in its corresponding table. At this moment, the classes still have not any operation. In the same way, the Relationship has a reference to both classes involved and keeps information about cardinalities and navigability.
In the case of using EJBs, the forward engineering stage implies the construction and code generation of the four afore-mentioned classes and interfaces; these ones are put into the domain tier of the application. For example, the classes and interfaces generated for the Person table are those of Fig. 9. By default, the Home interface includes the standard methods create (with so many parameters as columns the table has conforming its primary key), findAll and findByPrimaryKey. If the table is referencing any other table in the database, the corresponding Home interface also includes a method to locate instances by navigation (for example, the CarHome interface includes the method findByPerson(String Name, String LastName) to recover the collection of cards of a given person. The Home interface includes also one or more methods to show results in HTML format.

The Remote interface (Person, in Fig. 9) includes the setter and getter methods (one for each field), as well as one or more method to recover data about the instance in HTML format. As it is known, the execution of the CRUD operations is directly managed by the component container; for this, the tool adds the EJB class the corresponding ejbLoad, ejbCreate, ejbStore and ejbRemove methods.
Fig. 9. Some classes for the business tier when EJBs are being used

If the final application will not be based on EJBs, but in standard Java classes, then two classes are generated in the domain tier for each table: a pure domain class containing the CRUD operations and the *getter* and *setter* methods, and an additional class (a Pure Fabrication pattern [10]) to recover results in HTML format and to allow the navigation with the related classes (Fig. 10).

Fig. 10. Some classes for the business tier when standard java classes are being used

In the presentation tier, for each original class/table, some JSP pages are created in order to manipulate the table records. As a difference, these pages are not represented in any metamodel, being generated according to a set of template files.
So, a JSP page called \textit{listX} is generated to recover lists of records of the \textit{X} table (left upper side of Fig. 11), and a\textit{formX} page to manage instances of the \textit{X} class. Those forms corresponding to tables referencing other table include buttons to recover the required field values of the referenced table, as it is shown in the bottom side of Fig. 11.

Depending on whether EJB or standard Java classes have been used in the domain tier of the application, there are small differences between the code of the JSP pages generated (Table 1).

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
With EJBs & With Standard Java classes \\
\hline
\end{tabular}
\caption{Different pieces of Java code embedded in the JSP pages}
\end{table}

3.3 Overview if the complete process

The general reengineering process is depicted in Fig. 12: one of the concrete factories extracts the database structure, obtaining a \textit{Database} object, what corresponds to the metamodel of Fig. 5. Metamodel for class diagrams. Then, a converter takes the database representation and obtains an instance of the Class metamodel (Fig. 5). This one is used by the suitable code generators.
4 Conclusions and Future Work

This article has presented some internal details of a tool for code generation from databases. The tool has already been used in several projects. In these ones, the adequate distribution of responsibilities across tiers and classes allows that, for an usual management information system, only about the 3% of the lines of code must be written by hand. We have made a maintainability analysis of the code generated that allows the prediction of some code metrics before the application generation. The following equations allow to predict the size, weighted methods per class and coupling between objects of each EJB, when this option has been selected to generate the application ($K_{LOC}$ and $W_{MC}$ are constant values representing the minimal size and complexity of each component). Usually, all values are under the thresholds established by [11].

$$LOC = K_{LOC} + 13 * N^o Of Indexes + 8 * N^o Col + 3 * N^o ColFK$$

$$WMC = K_{WMC} + 4 * N^o Col + 2 * N^o ColFK$$

$$CBO_{EJB} = \sum_{i=0}^{FK_i} N^o Cols(FK_i)$$

The tool architecture is very easily expandable: the implementation of a new factory to generate code from other database vendor takes no more than one hour. We have no required to generate applications for other target environments than Java-based, but the implementation of new code generators would be also quite easy.

We are now defining a new metamodel to represent state machines, in order to describe operations that could be added to domain classes, and adequately called from the presentation tier. To improve the functionality of the tool, we are developing a graphical environment to represent the object oriented system obtained from the database in order to customize the set of classes or EJBs obtained.

Currently, the tool does not adequately generates code for user-defined data types at the presentation tier, and this is another of our lines of work.

Relational databases are complex environments with many elements, so we have planned the improvement of the actual metamodels to support all the elements of the relational databases, as well as its migration to the object-relational paradigm. This last topic will not be very difficult to achieve due to the nature of the obtained system and the architecture of the application based on metamodels and factories.
5 Acknowledgments

This work is part of the MÁS project (Mantenimiento Ágil del Software), Ministerio de Ciencia y Tecnología/FEDER, TIC2003-02737-C02-02.

6 References

Good practices for creating correct, clear and efficient OCL specifications

Dan CHIOREAN, Dyan CORUTIU, Maria BORTES, Ioana CHIOREAN

“Babes-Bolyai” University - Computer Science Research Laboratory
Str. M. Kogalniceanu, 1
400084 Cluj-Napoca - Romania
chiorean@cs.ubbcluj.ro

Abstract. From its first official version 1.1, the UML static semantics was specified using the Object Constraint Language (shortly OCL). Unfortunately, the official documents describing the UML standard [10] contain many erroneous OCL specifications. A part of the OCL specifications are difficult to understand. This paper presents a set of recommendations supporting the modeler in realizing correct, clear and efficient OCL specifications. All the examples were tested on different UML models, using the OCLE tool. The utility of having correct and clear Well Formedness Rules (shortly WFR) goes beyond the opportunity of checking the compliance of UML models with the static semantics of the modeling language. Applying these good practices users can realize clearer and correct OCL specifications in an efficient manner. In the MDA context, this represents a valuable support in working with correct MOF models.

1 Why correct, clear and efficient OCL specifications? Why the examples are from the UML static semantics?

The first issue that needs to be made clear is what do we mean by correct, clear and efficient specifications. Also we will try to explain why specifications need to accomplish these three characteristics.

Correct specifications are those specifications that are entirely compliant with the objectives for which they were created. The semantic of correctness goes beyond the concordance between informal specifications (expressed in natural language) and formal specifications (expressed in a formal language). Correctness can be achieved through an iterative process in which the formal specification helps in improving, completing and reformulating the informal specification. Correctness represents the most important quality of a specification.

The clarity of specifications refers to the degree of their comprehensiveness. Clarity is essential in communicating specifications. Clearer specifications are also easier to debug. For medium or large specifications the amount of time needed for improving their correctness is substantially reduced.

Efficient specifications are those specifications that do not contain redundant operations that cause a significant increase of the evaluation time. Practice has proven that the evaluation time becomes an important parameter. The evaluation of a set of

---

1 A UML CASE tool designed and implemented at the UBB Computer Science Research Laboratory in Cluj Napoca, Romania – http://lci.cs.ubbcluj.ro/ocle
rules for a medium or large size model (that contains tens of thousands model elements), implies the execution of hundreds of thousands invariant evaluations. Obviously, in case of large models the importance of the efficiency factor increases. Even for small models efficiency can be important, because the automatic source code generation for inefficient OCL specifications leads to inefficient source code.

Another aspect that needs clarification addresses the source of our examples. All examples used are included in the UML specification, the signaled errors being present in all UML adopted versions until now (UML 1.1 to UML 1.5). What characterizes these specifications? The UML metamodel is object-oriented. This metamodel is a medium size model, with many dependencies between its model elements. The static semantics of UML is expressed in medium or large OCL specifications. Additional Operations (declarations introduced by def-let mechanism) are used extensively for invariant specifications. OCL specifications represent the exclusive way for formally specifying the static semantics of UML. Consequently, they have to be correct, intelligible and efficient.

2 Introduction – the trap of being superficial

Superficiality in using OCL and the lack of appropriate tools are, in our opinion, the main drawbacks that influenced in a negative manner the widespread use of OCL specifications. Regrettably, the OCL specifications defining the UML static semantics contain a lot of errors. This offer OCL detractors “examples” that prove, in their opinion, that the constraint language does not accomplish the requirements for which it was conceived. Using these “examples” as arguments against OCL, people fall in the trap of using and judging the specification language in a superficial manner. The paper “Toward Executable UML” signed by Scot W. Ambler in Software Development Magazine, January 2002 [14], fits perfectly the “pattern” mentioned above.

“Throughout the IT industry, OCL use is negligible—perhaps only one in 100 developers could tell me what OCL stands for, and of those people, only one in 100 could write an OCL statement. I rarely see OCL in client sites’ models, and when I do, it’s minimal...Developers quickly adopt the techniques and technologies that offer real-world value—unfortunately, this is what OCL clearly lacks.”

To support his statements, Ambler presents the example below, stating:

“The following code sample presents a well-formedness rule for associations in UML diagrams, taken from OMG document ad/01-08-35: "Initial Submission to OMG RFP's: ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL)". The elements of the UML metamodel used in this WFR are represented in Figure 3.

context Association_inv
  AllAssociationEndsHaveDistinctNames

“This code sample reveals the reason that OCL hasn’t been widely adopted: It's wordy and hard to read. In plain English, the rule simply means that all association ends in an association's namespace have different names—but you must write six lines of OCL code to represent that rule.”[14]
We must highlight that the specification cited by Mr. Ambler does not conform to the OMG specification. Mr. Ambler’s OCL specification contains many syntactic errors: the context declaration is incorrect, in the self context there are no features named AssociationEnds, in the AssociationEnds context there are no functions named name(). The specification is unclear and unnecessarily large. A correct and clearer specification is:

context Association
inv uniqueNames:
   -- All associationEnds connected to an association have distinct names
      self.connection->isUnique(ae | ae.name)

Considering that the invariant name is optional and that the use of self is optional, the formal specification is shorter than the informal one. Furthermore, the informal specification mentioned by Mr. Ambler is ambiguous. In Mr. Ambler’s informal specification the “association's namespace” is mentioned erroneously, changing completely the rule’s semantics. The namespace associated to an association must be at least in our opinion the “smallest” namespace including all the associationEnds connected to the association. Because Mr. Ambler did not mention anything about the association referred by self and the connected associationEnds, we have to take into account all the associationEnds defined in the “association's namespace”. Therefore, the OCL specification complying with Mr. Ambler’s informal specification is:

context Association
inv a_sintactically_correct_OCL_spec_but_semantically_unjustified:
   self.namespace.ownedElement-
   >select(oclIsTypeOf(AssociationEnd))-
   >isUnique(ae | ae.name)

Semantically, this rule is illogical because it forbids many legal situations encountered in design and in programming. Mr. Ambler’s example reminds us that having an appropriate specification language represents a necessary but insufficient precondition in obtaining a correct and understandable specification. Following some good specification practices is thus strongly recommended. In this paper, we will try to present a set of good guidelines and to prove that they are natural, simple and easy to achieve.

---

2 The UML specification did not mention anything about the namespace associated to an association.
3 State of the art

Recently, many good books included OCL among their main topics ([5], [6], [7], [9], [11], [12] and the list is not complete). A lot of papers were also published on this topic [3], [4], [8], [13] etc. However, the quality of OCL specifications and the requirements that must be fulfilled in order to obtain good specifications was quite neglected. As a result, a part of the OCL specifications contains errors or is difficult to understand. Unfortunately, as we mentioned in the previous section, the UML static semantics contains a lot of counter-examples of how OCL specifications must be done. In order to contribute to a widespread use of OCL, it is very important to eliminate all these errors. The quality of the Additional Operations and of the Well Formedness Rules formal specification has to be a strong argument in favor of OCL.

This goal can be achieved by using appropriate OCL tools and by considering the rules discussed in this paper.

4 Good practices to be considered in realizing OCL specifications

All the good practices mentioned below proved their efficiency and validity in other specification or programming languages. Ignoring these recommendations can lead to incorrect and inefficient specifications. Even in case of simple and intuitive languages like OCL, these practices should be considered.

- The informal specification has to precede the formal one. The informal specification must be as complete and unambiguous as possible. OCL specifications support an incremental and iterative process helping in obtaining fully compliant informal and formal specifications. Explaining the objective of the specification in its informal description proves helpful in developing the formal specification.
- The OCL specification must consider the object-oriented architecture of the UML model, including inheritance and operation redefinition.
- OCL is a simple and intuitive language. This does not automatically ensure the comprehensiveness of OCL specifications. The clearness represents one of the most important features of the OCL specifications. It deserves a special attention.
- OCL specifications have to be evaluated using an appropriate tool. The evaluation represents the only valuable way of validating OCL specifications. The validation of an OCL specification requires its evaluation on a significant number of UML models. The fact that a rule catches some inconsistencies does not imply that the rule is correct and complete.
- In order to become an industrial standard, the OCL specifications must be efficiently evaluated in case of medium and large size models. The efficiency concerns the amount of time needed to perform a large number of evaluation operations. Therefore, OCL specifications implying redundant computations are inefficient, and must be avoided.
5 Examples illustrating the consequences of ignoring the above mentioned recommendations

5.1 The informal specification must be as complete and unambiguous as possible

Among the Well-formedness Rules provided for the Core package, we have the following:

```
context DataType
inv WFR_2_DataType: -- [10] page. 2-58
  -- [2] A DataType cannot contain any other ModelElements.
     self.allContents->isEmpty

Figure 1 – The ownedElement in a Namespace – UML 1.5
```

As we may notice analyzing Figure 1 and the specifications of the allContents AO, both the informal and formal specifications are very simple but erroneous. The correct specifications are:

```
context DataType
inv WFR_2_DataType: -- [2] The Namespace defined by a DataType, cannot contain any
  -- ModelElement.
  self.ownedElement->isEmpty
```
The allContents definition:

```java
class Namespace {
  def allContents:
    -- The operation allContents results in a Set containing all
    -- ModelElements contained by the Namespace
    let allContents: Set(ModelElement) = contents
    let contents: Set(ModelElement) =
        self.ownedElement-
        >union(self.namespace.contents)
}
```

highlights that, for this AO, the formal specification does not comply with the informal one. All the elements contained in a Namespace (with the exception of the Classifiers’ Features) can be obtained by navigating the association (composition) between Namespace and ModelElement. Due to their recursive definition, the contents and allContents AO contain all the elements defined in the Namespace referred by self, joined with all the elements defined in the Namespaces including the Namespace referred by self.

5.2 The OCL specification must consider the architecture of the UML model

Let us analyze now the Additional Operation named allConnections defined in the AssociationClass metaclass. As the informal specification mentions, this operation computes “the set of all AssociationEnds of the AssociationClass including all connections defined by its parents”. For example, in case of the model represented in Figure 2, the result obtained after evaluating allConnections AO must be:

```java
class C {
  def allConnections:
    let allConnections:Set(AssociationEnd) = Set{a, b, cb, da, e, f}
}```
The explanation is simple: an AssociationClass can inherit from another Classifiers like Class or AssociationClass. Therefore, when we compute the parents’ connections we must consider the type of the parent.

The OMG specification did not consider the particularities mentioned above. Furthermore, this specification contains compilation errors, marked here by underlining the erroneous sub-expressions.

context AssociationClass
let allConnections: Set(AssociationEnd) =
self.connection->union(self.parent->select(s | s.oclIsKindOf(Association))->collect(a: Association| a.allConnections))->asSet

This “kind of” error was identified in many AO and WFR. The first error concerns the type returned by the select operation – this is always the type of the collection on which we apply the select operation.

self.parent->select(oclIsKindOf(Association)):Set(GeneralizableElement).

In UML, parent always returns a set of GeneralizableElements. To obtain a set of AssociationClass enabling us to apply the allConnections operation, a cast must be performed.

The second error concerns the union operation. This can be performed if and only if both collections used in the operation comply with the collection type conformance rule. As each collect operation returns a Bag, and self.connection is of type Sequence(AssociationEnd), we have to change the type of one collection. Fixing these problems we obtain:

let allConnections: Set(AssociationEnd) =
(self.connection->asBag->
union(self.parent
>select(oclIsKindOf(Association)).
oclAsType(Association).allConnections))->asSet

Figure 2 – The UML user model used in evaluating the allConnections AO
This specification did not contain compilation errors, but the results obtained evaluating it on different models are not correct. In the context of the AssociationClass we obtain:

\[
\text{allConnections:} \text{Set(AssociationEnd)} = \text{Set\{a, b, e, f\}}
\]

The specification we propose is:

\[\begin{align*}
\text{context AssociationClass} \\
\text{def allConnections:} \\
\text{-- both the connections of the AssociationClass and the} \\
\text{-- oppositeAssociationEnds are considered} \\
\text{let connections:} \text{Set(AssociationEnd)} = \text{self.connection->asSet->} \\
\text{union(self.oclAsType(Classifier).oppositeAssociationEnds)} \\
\text{let allConnections:} \text{Set(AssociationEnd)} = \text{self.connections->} \\
\text{union(self.allParents->collect(p |} \\
\text{if p.oclIsTypeOf(AssociationClass) then} \\
\text{p.oclAsType(AssociationClass).connections} \\
\text{else p.oclAsType(Classifier).oppositeAssociationEnds} \\
\text{endif)->asSet)} \\
\text{select(e | e.visibility = #public or e.visibility = #protected)}
\end{align*}\]

This takes into account the architecture of the UML metamodel.
5.3 The OCL specifications must be as clear as possible

In order to exemplify this rule, we will analyze one of the most used AO defined in the Classifier context, oppositeAssociationEnds.

```
context Classifier

   oppositeAssociationEnds: Set(AssociationEnd); oppositeAssociationEnds = self.associations->select(a| a.connection-> select(ae| ae.participant = self).size = 1)->collect(a| a.connection-> select(ae| ae.participant <> self))-union(self.associations-> select(a| a.connection->select(ae| ae.participant = self).size > 1)-> collect(a| a.connection))
```

This specification has three compilation errors. Two are due to the incorrect use of the “.” operator instead of “->”. The last is because the type computed for the specification, Bag, did not match with the declared type, Set.

The main drawback of the above specification is its lack of understandability. To obtain a clearer specification, we will analyze how Classifier,
AssociationEnd and Association are modeled in the UML 1.5 metamodel (Figure 3).

The operation oppositeAssociationEnds is computed easily, by adding to the resulting set the opposite associationEnds (connection) corresponding to each Classifier’s association. In case of auto-associations, both elements of the connection (see Figure 4) must be included.

The AO we propose implements the algorithm mentioned above. A new function acOppAssEnd(as:Association, c:Classifier) is used to compute the opposite associationEnds corresponding to the Association as, connected to the Classifier c.

```plaintext
class Context Classifier

  def oppositeAssociationEnds:

  -- The opposite associationEnds of an association-classifier pair

  let acOppAssEnd(as:Association, c:Classifier): Set(AssociationEnd) =

      (if as.connection.participant->isUnique(e| e) -- association ?
       then as.connection->select(ae| c.association->excludes(ae))
       else as.connection -- auto-association
           endif)->asSet

  let oppositeAssociationEnds: Set(AssociationEnd) =
      self.associations->iterate(ass;
      acc:Set(AssociationEnd) =
          Set{}| acc->union(acOppAssEnd (ass, self)))
```

Evaluating this AO in case of Person, a Classifier instance represented in Figure 4, we will obtain:

oppositeAssociationEnds: Set(AssociationEnd) =
Set{ employer, job, wife, husband, marriage, bank, managedCompanies }

Figure 4 UML user model used in evaluating the oppositeAssociationEnds AO

5.4 Evaluation represents a mandatory step in validating OCL specifications – recursive specifications need a careful testing strategy

Next, we will present two OCL specifications that cannot be evaluated. First we will study the Additional Operation allSuppliers.

category ModelElement
def allSuppliers:
-- [2] The operation allSuppliers results in a Set containing all the
-- ModelElements that are suppliers of this ModelElement, including
-- the suppliers of these Model Elements. This is the transitive closure.

let allSuppliers: Set(ModelElement) =
  self.supplier->union(self.supplier.allSuppliers)

In this case, the OCL specification is simple and suggestive. This specification cannot be evaluated if there are circular dependencies among the model elements, like in Figure 5.
In order to solve the problem, in a manner conforming with the simplicity and intuitiveness of the OCL language, OCLE defines an additional operation on OCL collections that computes the transitive closure. Using this new operation, the allSuppliers specification becomes:

```ocl
def allSuppliers: Set(ModelElement) = Set(self)->closure(it| it.suppliers)
```

Another erroneous recursive specification was done for the contents Additional Operation, defined in the Namespace context:

```ocl
def contents: Set(ModelElement) = self.ownedElement- >union(self.namespace.contents)
```

Considering that the namespace associated to the Model metaclass instance is undefined and the recursive nature of this specification, every evaluation will fail, returning undefined.

To eliminate this mistake, we propose the following specification:

```ocl
def contents: Set(ModelElement) = if self.namespace->isUndefined
 then self.ownedElement
 else self.ownedElement- >union(self.namespace.contents)
endif
```

This is also useful for the automatic translation of OCL specifications in programming language code, so that the resulting program behaves correctly when executed.
5.5 The OCL specifications implying redundant computations are inefficient, and must be avoided

The first example

context Namespace
inv WFR_1_Namespace:
  -- If a contained element, which is not an Association or Generalization
  -- has a name, then the name must be unique in the Namespace.

    self.allContents->forAll(me1, me2: ModelElement| (not
      me1.oclIsKindOf(Association) and
      me2.oclIsKindOf(Association) and
      me1.name <> "" and me2.name <> "" and me1.name =
      me2.name)
    implies me1 = me2 )

Apparently simple and clear, the above specification contains some drawbacks:

- In the Namespace context, the AO specification is: allContents: Set(ModelElement) = self.contents. Therefore, in the above rule, all the elements defined in the analyzed namespace and in all namespaces including it are considered. When the model elements are used in a namespace different from the namespace where these elements were defined, the model elements names must be prefixed with the path, (formed by concatenating the namespaces' names separated by "::"). This naming rule changes the name uniqueness requirement from the model level to the namespace level. Considering the naming rule, allContents must be replaced by ownedElement. This way, when evaluating the rule, only the name conflicts concerning the model elements defined in the namespace analyzed will be reported and the number of operations performed in the rule evaluation process will be reduced.

- Another problem can appear when the rule is evaluated for namespaces containing at least two AssociationClass instances, both having the same name. This conflict will not be reported by this rule, because the expression me.oclIsKindOf(Association) is evaluated to true, for each me:AssociationClass. This problem can be solved by replacing oclIsKindOf(Association) with oclIsTypeOf(Association).

- Another problem is due to the fact that in the formal specification, the Generalization instances were not excluded as stated in the informal specification. The rejection of unnamed elements cannot guarantee the rejection of all Generalization instances.

- Finally, the efficiency problem. Evaluating the rule in case of didactic examples the specification below was evaluated faster compared with the OMG specification. In the OMG specification, the elements rejection was included in the forAll operation. As a result, $n^2$ operations will be performed for each oclIsKindOf
and for the “=” operation. In the example below, the above-mentioned operations will be performed n times.

context Namespace
  inv WFR_1_NAMESPACE:
    self.ownedElement->reject(e| e.name="" or e.oclIsKindOf(Association)
    or e.oclIsKindOf(Generalization))
    >isUnique(e| e.name)

The second example

context Class
  inv WFR_1_CLASS:
    -- If a Class is concrete, all the Operations of the Class should have
    -- a realizing Method in the full descriptor.
    not self.isAbstract implies self.allOperations->
    forall(op| self.allMethods->exists(m| m.specification-
        >includes(op)))

As we can notice, the value of self.allMethods was included in the body of forall operation, being pointlessly computed for each operation. Using the let construction the time of evaluation will diminish – allMethods.specification will be evaluated only once.

context Class
  inv WFR_1_CLASS:
    let implMet:Bag(Operation) = self.allMethods.specification
    in not self.isAbstract implies self.allOperations-
    >forall(op| implMet->includes(op))

Considering the informal specification and the UML model, a simpler and clearer specification is:

    not self.isAbstract implies
      self.allMethods.specification->asSet = self.allOperations

6 Conclusions

The main purpose of this paper has been to present a set of recommendations supporting correct, clear and efficient OCL specifications. The effect of these guidelines was analyzed by means of the OCL specifications made at the UML metamodel level. A significant number of errors were discovered in the specification of both the Additional Operations and the Well Formedness Rules. New specifications were proposed for the
UML static semantics. These are clearer and more efficient compared with the existent specifications and overcome the drawbacks found in UML standard. We were surprised to notice that the errors signaled in [8] were not taken into account in the OMG specifications.

The conclusions we reached can be summarized as follows:
- The static semantics is specified using two different formalisms: an informal (the textual specification) and a formal one (the OCL specification).
- The formal specification translates the informal specification by using a rigorous language.
- The OCL language proved to be a good support in identifying ambiguous or incomplete textual specifications.
- In order to validate OCL specifications, it is mandatory to evaluate them. The fact that the specifications had no compilation errors didn’t suffice.
- Each specification has to be tested both on correct and incorrect models.
- The UML Metamodel architecture is object-oriented. Consequently, both the Additional Operations and the Well Formedness Rules have to take into account the object-oriented and the design by contract principles.
- Considering the above statement, it is strongly recommended to mention for each specification: the contexts where it is used, the contexts where the specification is overwritten.
- For each informal specification it is possible to have different formal specifications. Considering that OCL is a powerful specification language, simple to understand, it is extremely important that the formal specification is as simple and clear as possible.
- The OCL specifications given at the UML metamodel level are by far more complex than the OCL specifications made at the model level [5], [6], [11], [12].
- The def – let mechanism represents a powerful instrument, useful in managing large OCL specifications. It supports an easy extension of the UML API.
- Apart from intuitiveness, good OCL specifications are optimized from the point of view of the evaluation time.

We hope that the conclusions presented above and the solutions we proposed will support the UML community (including OMG) in writing a correct static semantics for UML 2.0 in a short time.

References


Performance Modeling and Visualization with UML

Juha Gustafsson

Department of Computer Science
P.O. Box 26, FIN–00014 University of Helsinki
Finland
Juha.Gustafsson@cs.helsinki.fi

Abstract  UML is the de facto standard of object-oriented software modeling. Thanks to being based on well-known conventional diagramming techniques, it is easy to learn and yet very versatile. It is, however, not so commonly used as a performance engineering tool. New profiles and extensions to the UML standard itself have improved this situation. We discuss the issues we need to take into account when using UML diagrams for performance evaluation as well as some of our experiments.

1 Introduction

Maisa (Metrics for Analysis and Improvement of Software Architectures) is an ongoing research project for developing methods and tools for automatically analyzing the quality of a software architecture [3,7,15]. The Maisa approach is based on calculating software metrics as well as searching for architectural and design patterns in the architecture, all expressed as UML diagrams. The extracted information is then used for evaluating the quality of the software in question. Among the quality factors that are examined, size, complexity, and run-time performance are the most important.

To evaluate software performance, we need a software execution model. The basic UML, however, lacks this kind of modeling technique [1,2,8]. The emphasis in the UML diagrams that model dynamic behavior can be on the flow of control (i.e. the activities) or on the objects that interact. For performance modeling we prefer to emphasize the activities.

Class or object diagrams can be used to estimate the run-time memory requirement of the software [6]. The basic procedure is the following. We select one class as a basis class. Then we estimate the number of instances of other classes created relative to a single instance of the basis class. When we have an approximation of the number of instances, we use the size information of each class to calculate the total run-time memory requirement. For a more accurate estimate, we would have to know which instances exist at the same time (active instances). Then, sequence diagrams can be used to estimate the number of those instances.

Real-time system models often have very specific time or space constraints. Critical issues are the latency (the delay) and the duration of a message. The Maisa tool will notify the user if some of the constraints are not satisfied. Since the performance estimate may be rough, the user must specify a tolerance for each constraint. The tolerances play an important role as the lack of design information in the early phases makes any predictions less accurate. The Maisa tool can help to estimate both execution time and run-time memory requirements of the modeled program.

We proceed as follows, in Chapter 2 we discuss UML state charts and activity diagrams from the performance point of view. We then concentrate on our extended activity diagrams and using them in execution time and memory usage estimation in Chapter 3.
In Chapter 4 we discuss a related performance modeling technique, the execution graph model, as a comparison to the UML based methods. Chapter 5 presents a summary and a brief outline of our future work.

2 State charts

State charts have long been used to describe the life cycle and reactive behavior of a system [4]. They have a solid theoretical base from finite state machines. They have good expressive power with nested substates and orthogonal components. State charts can even include conditional connectors, timeout transitions and highly complex action lists.

Although state charts are used to describe functionality, they usually have a very restricted scope, typically they model the internal actions of objects in a single class. They model the behavior (object states, transitions and their respective triggers) of objects rather than the actual flow of control through the system. The transitions usually take place in response to outside events rather than a completed activity.

For the Maisa tool state charts would provide the necessary means for modeling the behavior of a system. To evaluate the performance of a system, we need a complete description of its flow of control. It would be impossible (or at least very tedious) to get this information from smaller scale diagrams. Thus, we would like to emphasize the global flow of control rather than object-specific events. This leads us to activity diagrams, that are extended state charts themselves.

2.1 Activity diagrams

Activity diagrams are an extension of state charts and as such they have the same expressive power. Unlike in state charts, the nodes (called activity states) in activity diagrams don’t have to be within the same object. Decision points allow the user to represent branching in the control flow.

The advantage of activity diagrams is that they are already included in UML and in many CASE tools that are capable of producing UML diagrams. Activity diagrams of standard UML support concurrent modeling, but resource information must be added to them to support performance modeling and analysis. They model the flow of control well, but they do not have adequate representation for resource usage.

We need to use both timing marks (when does a particular activity begin) and timing constraints (how long it can last) as well as timing information given by the user (how long it will last). These have to be specifically defined (using e.g. the Object Constraint Language [5]).

3 Performance analysis in Maisa

Performance analysis requires detailed access to the parameters of a particular diagram. At least timing information and the probabilities attached to transitions must be modifiable within Maisa. Additionally, the structure and layout of the diagrams is critical. Regular metrics, even pattern mining results can be adequately represented using only a textual format, but for performance analysis, a more visual environment is required.

Extended activity diagrams have been selected as a modeling tool for performance analysis in the Maisa project. They are familiar enough for anyone accustomed with UML diagrams, while the additional information enables their use for this purpose. Figure 1 shows an example extended activity diagram.
3.1 Nodes

When using activity diagrams for performance evaluation, we need to add timing information to the basic UML activity states. In our model each node contains (at least) three time attributes: the minimum, typical, and maximum execution times of the node.

In order to support stepwise refinement and information hiding, we create hierarchical models by allowing activity nodes to be expanded as further activity diagrams. This kind of hierarchical structure is vital for any practical application. The depth of the hierarchy and the appropriate level of detail depend on the current design stage as well as the user’s needs. For more accurate estimates, we need a more detailed diagram.

If a node is expanded (i.e., has a subdiagram), then we obtain all time information from the subdiagram. This means that we always calculate first the subdiagrams. Graphically we denote an expanded node with a special node symbol (see Figure 1). The timing information for all simple nodes is shown as a triplet \[ \text{minimum} - \text{typical} - \text{maximum} \]. If an expanded node contains explicit minimum and maximum times, these can be used as constraints for the values obtained from the subdiagram.

As with regular activity diagrams, we have several special types of nodes. Each diagram (or substate) has a single start-node and one or more stop nodes. Decision points contain a condition according to which the outgoing transition is chosen.

Another special type of node is the loop node. A loop node begins a loop. We denote a loop node with a rectangle around the ellipse. In addition to the timing values, it contains the minimum, typical, and maximum number of repetitions. These limits are shown along the lower edge of the rectangle. Note that a loop node cannot be expanded.

3.2 Transitions

The transitions are very much like those in common activity diagrams. Each transition may contain a probability (how likely it is that the respective condition is triggered).

Loops are allowed, but they must always be accessed via their start nodes. Thus all transitions to a node inside a loop originate from nodes that are inside the same loop.
However, any node inside a loop may contain transitions that lead outside the loop. Any end node (i.e. a node that has a transition to the beginning of the loop) may have one or more transitions that lead out of the loop.

### 3.3 Time analysis

When we use the extended activity diagrams for time analysis, we begin with a set of nodes and a set of successors for each node (i.e. the nodes to which there is a transition from this node). We want to find a path from a given node (the start node) to some end node. In our case we are interested in very specific paths (e.g. the one having the shortest execution time). We define these paths recursively. The ending condition of this recursion is an end node. Otherwise the paths originating in a given node contain the node combined with the paths from each of its successors.

Correspondingly, the time value of a path originating in a given node is the (appropriate) time value of the node itself combined with the values of its successors. The exact procedure depends on the metric needed. For the minimum time, we select that outgoing path, which gives the smallest value. For maximum time, the path producing the maximum value is selected. For typical time, we multiply the typical time value of each successor with the probability of the respective transition and add up the results. Figure 2 shows the result panel of Maisa.

The expanded nodes as well as loops are calculated separately. For loops we need to determine the nodes belonging to it (i.e. the nodes that have a path leading to the start node of the loop).

For easier overview, we have chosen to show the expanded nodes using the “rubber sheet” technique [9,12] (See Figure 3). The user may open one or more expanded nodes to view them in more detail.
3.4 Estimating memory use

For memory use estimation we need significantly more information than for timing analysis (See Figure 4). For each activity diagram, we need one or more class diagrams. Additionally, we need a sequence or collaboration diagram for each (non-special) activity node. We collect this information using a bottom-up approach to get an estimate for the whole scenario. Note that for existing components we may use real memory (or time) values if such are available.

For each sequence diagram, we know which objects have explicitly been created or destroyed. These (heap) objects form an object sequence, that describes how the use of heap memory varies during the execution of the sequence. When we combine this estimate with the effect of static and stack objects, we get a combined estimate. Because of the use of design level diagrams, these estimates are rough. However, Maisa can be used to discover interesting execution paths or nodes. For example, in which node the amount of used heap memory was at its maximum.

4 Execution graph model

The execution graph model [10,11,16] offers a relatively simple representation of software execution. As the model is based on mathematical graph theory, analysis algorithms can easily be applied to it. The model’s main advantage is that it has been developed specifically for performance modeling. Unfortunately, it is a technique entirely separate from UML. Thus, many conventional CASE tools don’t support execution graphs. Although it is possible to extend some of the tools to do so, one problem still remains: the developers would have to learn yet another technique.

Apart from notational differences, activity diagrams and execution graphs have much in common. They both have an activity-centered view to the system. The emphasis is on the flow from activity to activity rather than on objects that communicate with each other.

Several types of nodes in the execution graph are used to represent software components. One node can represent a few lines of code, a single method or even a set of
methods. As with activity diagrams, nodes can be decomposed into more detailed diagrams.

The arcs represent the flow of control through the model. In addition to the basic arcs, there are others that represent more complex situations. The call-return arcs, for example, represent control that returns to the calling node. If necessary, arcs from a case node can have branching probabilities.

![Diagram](image-url)

**Figure 4.** Diagrams required for memory estimation.

In the example in Figure 5, two players, A and B, play a game. Whoever wins ten sets first, wins the whole match. The node *initiate session* contains relevant operations for the gaming session (this can be specified further if necessary). The node *get input* reads the

![Diagram](image-url)

**Figure 5.** A two player gaming session as an execution graph.
players’ moves and calculates the winner of this set (this node could be expanded further). Depending on the winner of this set, we select the appropriate branch of the case node and perform the relevant operations. The repetition ends when either player has won ten sets.

For performance analysis, we have to specify the resources needed by each component (e.g. the execution time on a given node) and any additional information (e.g. the number of times the component is repeated). There are many parameters that can be varied, so this stage may require a lot of user interaction. Usually we don’t know the exact values for these parameters at this phase, but we can still try different values to get several estimates.

Execution graphs provide all the means for performance calculations. We would, however, need to implement them into our CASE tool. An alternative solution would be to use a separate program to construct these graphs and transform them into Maisa’s intermediate format. In addition to the fact that this would mean one more auxiliary program, it would decentralize the whole software engineering process.

5 Conclusion and future work

We have discussed different possibilities for performance modeling of a system on its architectural level and their advantages and disadvantages from the Maisa point of view. The key issues for any modeling technique to be used in performance evaluation are hierarchical representation of the flow of control and the possibility to include resource constraints such as probabilities and execution times. Any one of the diagram types discussed could be used for this purpose, but for the standard UML diagrams, specific resource information would have to be added.

Basic performance evaluation feature has been implemented in the Maisa tool. For this purpose, we have used extended activity diagrams, because they are already part of the UML and they have enough expressive power so that we can use them with little modifications. Because we only get rough estimates using design diagrams, they should be used as a starting point in performance improvement.

We are experimenting with using design pattern information to assist in detecting potentially performance critical components in the software. For example, several patterns designed for de-coupling or encapsulation purposes have a tendency to be poor from the performance point-of-view as they increase the number of method calls [14].

A new diagram type, the interaction overview diagram [13], has been proposed to the upcoming UML 2. These diagrams are very similar to the extended activity diagrams in that the nodes can represent another interaction diagram. The interaction overview diagram, as well as the other new diagrams types in UML 2 will be incorporated in Maisa as soon as possible.

Another important topic for further development is the comparison of other techniques such as execution graphs or linear programming [17] to the UML based, bottom-up method.

Acknowledgments

The Maisa development project has been funded by the Academy of Finland, the National Technology Agency of Finland and by four IT companies: Kone, Nokia Mobile Phones, Nokia Research Center, and Space Systems Finland. The Maisa tool is being developed as part of UML++, a collaborative research project of the universities of Helsinki and Tampere, and Tampere University of Technology. Other main contributors within the
Maisa team have been Jukka Paakki, Inkeri Verkamo, David Gregorio García, and Topias Marttila.

References

Towards the Unification of Patterns and Profiles in UML

Petri Selonen, Kai Koskimies, Tommi Mikkonen

Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{petri.selonen, kai.koskimies, tommi.mikkonen}@tut.fi
http://practise.cs.tut.fi

Abstract. Patterns have become a popular means to express recurring software solutions, as exemplified by design patterns. On the other hand, so-called profiles are used in UML to define various kinds of domain-specific architectural concepts and conventions as extensions of the UML metamodel. In this paper, we show how patterns and profiles can be unified in the UML context, using the UML metamodel as the common basis. We argue that this result has far-reaching implications on tool development, and helps us to understand the relationships of these two central software architecting concepts.

1 Introduction

The concept of a pattern has emerged as a response to the need to specify generic solutions to certain recurring software construction problems. In this context, a pattern is a description of an organization of abstract, system-independent software elements that provide a general solution to a problem in a context. A pattern can be instantiated in a particular system, which effectively binds the abstract software elements (so-called roles) of the pattern to concrete software elements in a particular system. Examples of patterns include analysis patterns [6], architectural patterns [1], design patterns [7] and coding patterns [3]. In a sense, a pattern can be regarded as a generic software unit comparable to, say, templates in C++, with roles corresponding to the template parameters. However, unlike templates a pattern is liberated from the primary decomposition of the system: a pattern can crosscut the components of the system in an arbitrary way.

In this paper we assume that a pattern is instantiated in a UML [10] model; thus, the roles of a pattern are bound to UML model elements. Unfortunately, standard UML provides only modest support for patterns: a special model element (collaboration symbol) can be used to mark an instance of a pattern in a model, with lines indicating the concrete model elements playing roles in the pattern. Consequently, efforts have been made to improve support for patterns in UML ([2], [4], [12]).

However, patterns are not always the most obvious vehicle for expressing architectural specifications. There is often a need to introduce various kinds of general conventions or rules, which are assumed to hold throughout the system. In particular, such rules are frequently imposed by the architectural decisions made in a particular software system, platform or domain. For example, a rule might state that because of a chosen architectural style, certain kinds of components may depend on each other only in a certain restricted, predefined way. In practice, such rules are typically enforced by coding standards given to system developers. In contrast to patterns, these rules are
system-wide, and their purpose is to preserve the architectural hygiene of the system rather than to offer a local solution to a specific problem.

In UML, a set of architectural rules can be presented as a so-called profile. A profile is an extension of the UML metamodel, specifying a subset of UML models that can be considered “legal” in a particular context (say, in a particular domain or software platform). In [13] a technique is presented for specifying a set of architectural rules as a UML profile, to be used for checking a given UML model against such rules. The characterization of a set of architectural rules as a UML profile (that is, as a metamodel extension) suggests that such a set of rules can in fact be viewed as a more restrictive modeling language the designer must follow. This is in sharp contrast with the notion of a pattern, which is seen as a specific arrangement of software elements that is voluntarily created by the designer.

In spite of their different character and purposes, patterns and profiles have much in common. Both are based on identifying certain model element categories, and on specifying relationships between these categories. In profiles a category consists of the instances of a new model element type added to the metamodel (so-called stereotype), while in patterns a category consists of the model elements playing a particular role in a pattern. In both cases, various kinds of semantic (or behavioral) information can be associated with the categories, originating from the problem context of a pattern or from the domain of a profile (like, say, the interaction protocol associated with a role or the domain-specific “meaning” of a stereotype).

Our thesis is that patterns and profiles can (and should) be unified. The main benefit of such a unification is that it would allow for a single tool that can be used to define patterns and profiles in UML modeling, supporting both the building of a model according to a pattern/profile and the checking of the conformance of a model with a pattern/profile. Moreover, we feel that it is important to find the common ground of software architecting concepts originating from rather different motivations, paving the way for conceptual convergence.

In this paper we take the first step towards the unification of patterns and profiles by demonstrating how a pattern can be interpreted as a profile and vice versa, assuming a particular (but reasonable) interpretation of these concepts. We outline the basic idea in terms of examples; we will not present a fully elaborated technique here but leave this for future work. In the scope of this paper, we will restrict ourselves to the structural properties of patterns and profiles, leaving behavioral issues out of the discussion.

The rest of this paper is structured as follows. Sections 2 and 3 introduce the concepts of profiles and patterns as we will treat them in this paper. Section 4 shows how to derive profiles from patterns, and Section 5 discusses how to create patterns out of profiles. Section 6 introduces some related work, and Section 7 concludes the paper with a discussion. Throughout the paper, we assume familiarity with UML.

1. UML and Profiles

UML has quickly emerged as lingua franca of software architecting. Although UML has been criticized for its lack of formal definition, its structural aspects are specified formally by the UML metamodel. This specification is given as UML class diagrams (strictly speaking, these class diagrams are given using the facilities provided by the

---

1 While the basic idea does not rely on a particular version of UML, we exploit some of the notations of UML 2.0 to facilitate the presentation.
metametamodel, including a subset of UML class diagram notation), associated with a set of well-formedness rules. From now on we will refer to these class diagrams as the UML metamodel. The classes appearing in the metamodel are called metaclasses, and the associations in the metamodel are called meta-associations.

As an example of the UML metamodel, Figure 1 depicts a simplified fragment of the metamodel, defining how classes, associations, and operations can be combined. Note that the metamodel requires that each association (class Association) has at least two association ends (class Property), and that each association end is attached to exactly one classifier (class Classifier, a generalization of class Class). Each class may in turn contain an arbitrary number of operations (class Operation). Two classifiers can be in a generalization relationship (class Generalization) with each other.

![Fig. 1. A fragment of the UML metamodel](image)

A profile is, by definition, an extension of the UML metamodel. Typically profiles are used to customize UML for a particular domain by introducing additional domain specific modeling concepts and constraints. The main mechanisms available for specifying profiles are stereotypes, constraints and tagged values. UML defines rather vaguely that a stereotype is “a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends” [10]. A stereotype can be added to the metamodel as a new class symbol which extends (a black-headed arrow, see Figure 2) an existing metaclass. Extension is a kind of an aggregation where the properties of the metaclass are extended through the stereotype, allowing a stereotype instance to be treated also as an instance of the metaclass without the stereotype.

The idea of stereotypes is to allow the introduction of slight extensions of the concepts of the metamodel, rather than entirely new kinds of modeling elements (which should be defined using MOF, the Meta Object Facility). Thus, a profile cannot change the existing properties of the model elements, but only add extended element types (stereotypes), their new properties (tagged values) and constraints. Consequently, tools can always manipulate instances of stereotypes as instances of their base element types, although some tools may exploit the additional properties defined for the stereotype.

Constraints can be given formally as OCL (Object Constraint Language) expressions over instances of the stereotypes, giving additional requirements for those instances. Here we will use constrained associations between stereotypes to express certain structural restrictions in a natural way. Strictly speaking, new associations are allowed in a profile only if the associations are subsets of existing meta-associations [10]. However, we use only additional associations that specialize an existing meta-association for a pair of stereotypes. Thus, the new associations do not introduce any new structural
relationships; they merely stand for existing relationships when applied to certain stereotypes. In our view, this does not conflict with the idea of a profile in UML, since any model that is an instance of the profile is also an instance of the metamodel. The profile only rules out certain configurations of (stereotyped) model elements.

Let us study the required structural constraints in more detail. Assume two stereotypes ST1 and ST2 have been introduced as extensions of two metaclasses MC1 and MC2 connected by a meta-association, as depicted in Figure 2a. With respect to allowed configurations of model elements there are four choices for structural constraints, given in Figure 2b: (i) there are no restrictions on the structural composition of the instances of the involved metaclasses and stereotypes beyond those implied by the metamodel, (ii) the structural relationship represented by the meta-association can exist for two metaclass instances or two stereotype instances, but not for mixed combinations, (iii) in addition to (ii), an instance of the first metaclass can be linked with an instance of the second stereotype, and (iv) in addition to (ii), an instance of the first stereotype can be linked with an instance of the second metaclass.

![Diagram](image)

**Fig. 2.** Profile specification and possible model element configurations

To express the structural constraints given above, we use an OCL template of the form:

```
template <metaclass MC, navigation NAV, stereotype ST>
context MC
inv:
    self.NAV ->foreach(stereotype->includes(ST))
```

The constraint template is instantiated (by replacing the template parameters MC, NAV, and ST2 with actual names) and attached to a stereotype appearing in a profile. Thus, the context of the constraint (that is, “self”) is actually an instance of a stereotype (extended from metaclass MC), since the constraint will be applied only to the instances of the stereotype. The constraint simply says that the model element at the NAV-end of a link must be an instance of stereotype ST.

---

2 Actually OCL does not allow templates, but we use here a C++-like template notation to facilitate the presentation.
We will name this constraint “strict”. We assume that the location of this constraint in the profile model determines the template parameters and thus instantiates a concrete form of the constraint with actual names taken from the profile model. Instead of attaching this constraint visually to a stereotype symbol itself, we attach it to an association end owned by that stereotype. In this way the template parameters get fixed: NAV will be the role name (or target class) at the other end of the association, ST is the stereotype whose instance should be found at that end of the link, and MC is the metaclass of the stereotype at the constraint end.

Using constraint \{strict\} in the way described above, we can now express a profile specification yielding one of the structural constraints (i) – (iv) above, as depicted in Figure 3. Informally, \{strict\} means that only this kind of model element (that is, a model element of the associated stereotype) can appear at this end of the link.

![Fig. 3. Specifying different structural constraints](image)

In cases (ii) – (iv), we require that the specialized association between stereotypes does not introduce new names for the association or its end roles. Further, we require that the multiplicities of the original meta-association are followed in the instantiation in all cases (i) – (iv). If several specialized associations share the same base meta-association, the sums of the multiplicities of the “subassociations” must be subsets of the multiplicities of the base meta-association.

In a model, an instance of stereotype S in a model is denoted by \(<<S>>\) preceding the name of the model element. Whenever such an instance appears in a model, we assume that the constraints given for the stereotype are checked. We also assume that only defined stereotypes are used in the model.

As an example, consider a situation where the architect has chosen a client-server architectural style, and wants to enforce this style in the design. A profile specifying the rules implied by this decision is given in Figure 4. The profile states that if a class stereotyped as \(<<Client>>\) is used in a model, there must be another class stereotyped as \(<<Server>>\), and there must be exactly one association between these classes stereotyped as \(<<service>>\). The profile states also that a \(<<service>>\) association may appear only between a \(<<Client>>\) and a \(<<Server>>\) (note the use of the \{strict\} constraint at the Client and Server ends of the associations). However, the profile allows
other, arbitrary associations between a <<Client>> and a <<Server>> as well, and associations between a <<Client>> (or a <<Server>>) and some other classes, since the ServerEnd and ClientEnd ends of the associations are not constrained.

![Diagram of UML profile specification for client-server architectural style](image)

Fig. 4. A UML profile specification for client-server architectural style

Checking a design model against architectural profiles such as the ones in Figure 4 has turned out to be useful in practice [13]. Often it makes sense to introduce a fairly large set of system or domain specific class categories as stereotypes, and specify the allowed dependencies between instances of these categories. If a model consists of hundreds of classes, which is often the case in real life, manual checking of the model would become very cumbersome.

### 2. Patterns

We define a pattern as an arrangement of software elements for solving a particular problem. Depending on the nature of the problem, we may speak of analysis patterns, architectural patterns, design patterns etc. In the sequel we will give a simple technical characterization of a generic pattern concept from the purely structural viewpoint. Thus, we ignore aspects like behavior or purpose which are of course important ingredients of, say, design patterns, but which are not an issue here.

To be able to define a pattern independently of any particular system, a pattern is defined in terms of element roles rather than concrete elements; a pattern is instantiated in a particular context by binding the roles to concrete elements. A role has a type, which determines the kind of software elements that can be bound to the role; the set of all the role types is called the domain of the pattern. Here we assume that the domain of a pattern is UML; that is, all the roles are bound to UML model elements.

Each role may have a set of constraints. Constraints are structural conditions that must be satisfied by the model element bound to a role. For example, a constraint of
association role \( P \) may require that the association bound to \( P \) must appear between the classes bound to certain other roles \( Q \) and \( R \). A containment constraint requires that the element bound to the role must be contained by the element bound to another role. For example, a containment constraint of operation role \( S \) may require that the operation bound to the role is contained by the class bound to class role \( T \).

A cardinality is defined for each role. The cardinality of a role gives the lower and upper limits for the number of the instances of the role in the pattern. For example, if an operation role has cardinality \( [0..1] \), the operation is optional in the pattern, because the lower limit is 0. The default cardinality is \( [1..1] \).

As an example of a pattern, consider Observer [7]. Slightly simplifying, the structure of this design pattern could be presented in UML as depicted in Figure 5. We assume there is a standard stereotype \(<\text{pattern}>\): a diagram (rectangle with a name in the upper left corner) stereotyped as \(<\text{pattern}>\) should be interpreted as a pattern specification rather than as part of a concrete model. The pattern interpretation of the diagram implies that every model element in the diagram represents a role; that is, a placeholder for a concrete model element. Thus, in Figure 5 there are roles for classes, operations, associations etc. A role can be unnamed, like the generalization (inheritance) relationship role. The type of a role is the metaclass of the corresponding model element. For example, the type of role “notify” is the metaclass Operation.

![Observer design pattern](image)

**Fig. 5.** Observer design pattern (simplified)

The class diagram in Figure 5 looks like an ordinary class diagram. The only place where we need additional notation (except for the \(<\text{pattern}>\) stereotype) is the specification of the cardinalities of roles. In the Observer pattern, we want to be able to provide several concrete observers derived from the same abstract observer. However, this is difficult to express naturally in UML: multiplicities have different meaning in UML. Therefore we specify the cardinality of a role (ConcrObserver) with a separate comment.

The pattern interpretation of a class diagram implies certain structural constraints for the roles:

- Each operation role has a containment constraint forcing the operation bound to the role to appear within the class bound to the enclosing class role.
- An inheritance role appearing between two class roles has a constraint requiring that the classes appearing at the ends of an association bound to this role must be bound to the two class roles.
Similarly, if an association role appears between two class roles, the association role has a constraint requiring that this role must be bound to an association between classes bound to the class roles.

- If a class role inherits another class role, and both class roles contain an operation role with the same name, the operation role in the inheriting class role has a constraint requiring that the operation bound to this role must have the same signature as the operation bound to the role contained by the superclass role.

In the example, the syntactical consequences of these constraints are that we require that “update” operations can appear only in classes playing the Observer and ConcrObserver roles, and that “register” and “notify” operations appear only in a class playing the Subject role. Further, “notifies” association may appear only between classes playing the Subject and Observer roles, and “registers” association only between classes playing the Subject and ConcrObserver roles. In each instance of the pattern, there can be an arbitrary number of classes playing the ConcrObserver role. Finally, each class playing the ConcrObserver role must inherit a class playing the Observer role.

This kind of structural pattern concept can be used as a basis of fairly powerful tool support for pattern-driven software development. In [8], the general idea is that the specialization interface of an application framework (in Java) is described using such patterns, with some of the roles bound to the framework’s elements. The tool is able to maintain a task list for binding the application dependent roles. The task list displays a task for each role that can be instantiated and bound in the present situation, taking into account the dependencies implied by the constraints and the multiplicities of the roles. The tool keeps track of the constraints and generates additional tasks to correct constraint violations. The patterns can be augmented with practically useful information, like templates for default elements to be bound to a role (allowing the generation of the bound element), intuitive task prompts, informal explanations of the tasks etc. Recently, the tool has been extended to support UML based design in Rational Rose [11]. The patterns, however, are specified in that tool using a dedicated graphical presentation rather than UML.

In the following two sections, we shift the focus to the actual unification of profiles and patterns, and show how to translate patterns to profiles and vice versa using UML metamodel as the common language.

### 3. From Patterns to Profiles

To understand the presentation of a pattern as a profile, consider again the pattern in Figure 5. Every model element in the diagram represents a role, which can be played by several model elements in an actual system model. Thus, each model element in the pattern represents a category of concrete model elements. This suggests that each model element (or role) in the pattern should be viewed as a stereotype. The base class of the stereotype is the element type of the role, e.g., for a class role, the base class is Class, for an association role, the base class is Association etc.

However, there are three kinds of model elements in the model of Figure 5: named explicit model elements (e.g., Subject), unnamed explicit model elements (e.g., the inheritance relationship), and implicit model elements (e.g., the association end elements of the associations). To really see all the model elements and their relationships we must construct the object representation of the model, according to the metamodel. This is shown in Figure 6. Note that the rectangles are now instances of the metaclasses (metaobjects), and the connecting lines are links (instances of associations).
To turn this pattern into a profile, we must transform the metaobject representation into an extension of the metamodel. This can be done in a relatively simple way. We already noted that each model element in the pattern should give rise to a stereotype. Thus, we introduce a stereotype for each metaobject appearing in the object representation of the pattern. The base class of the stereotype is the class of the metaobject, and the name of the stereotype is the name of the metaobject. To guarantee unique names of the stereotypes we consider class roles as namespaces for the contained roles (e.g., operation roles). The transformation rule is depicted in Figure 7.

Each link in the metaobject representation of the pattern is an instance of a distinct meta-association or its specialization. Thus, for each link there should be either a distinct, existing meta-association between the corresponding metaclasses, or a specialization of such an association between two stereotypes. The latter is needed only if the related pattern rule excludes some structural combinations that are allowed by the pure metamodel. In that case we need additional transformation rules to create the specialized (constrained) associations in the profile specification.

To allow the pattern provider to give the information concerning the structural constraints to be attached to the profile, we propose the following convention. A pattern model may include elements which are named as “any”. These could be, for instance, classes, operations, attributes, associations etc. The general rule is then that anything not explicitly allowed by the profile is forbidden. If the pattern provider wants to relax
certain structural constraints imposed by the pure pattern description, she can add “any” elements meaning that in any legal UML element can appear in the structural position of the “any” element, not just the particular kinds of elements denoted with a specific name (and later transformed into stereotypes).

Using this convention, the Observer pattern could be given as presented in Figure 8. This specification implies that both a subject and a concrete observer may have any other types of operations in addition to those specific to the pattern, and that these classes may also have any types of association with each other and with other classes, in addition to the given ones. However, the pattern does require that “notifies” and “registers” associations may appear only between the participants of the pattern in the specified way.

The additional information relaxing the strict interpretation of the structural constraints implied by a pattern description is of course out of the scope of the pattern concept itself: here a pattern description clearly gets some of the flavor of a profile. We argue that this is mostly a tool issue: a tool can hide this information, when a pure pattern view is desired. When the information is shown, it should be visually distinguishable from the rest of the pattern, as in Figure 8.

Assuming a pattern description following this convention, the metaobject representation will include the corresponding “any”-objects as well. Based on this, we can express the transformation rules producing the desired constraints in the profile. These rules are given in Figure 9. The crossed model elements denote missing elements.
As an example of the application of the transformation rules for associations, consider roles update (of Observer) and Observer. Their presentation in a profile is depicted in Figure 10.

Cardinalities of roles are somewhat trickier to transform into a profile representation, and we will not give any general rule here. The principle is illustrated by the example pattern, where role ConcrObserver has cardinality [0..*]. Cardinalities are given to class roles with respect to the relationships the role has to other class roles. In this case, ConcrObserver has an association relationship to Subject and an inheritance relationship to Observer. Thus, for each class playing the role Subject and for each class playing the role Observer, there can be an arbitrary number of classes playing the role ConcrObserver. Naturally, the relationship instances should be multiplied as well. Therefore we should locate the stereotypes representing the relationships, and attach the multiplicity to the associations leading to the stereotypes Subject and Observer in the profile.

Finally, additional conventions used in a pattern description have to be included in a profile in the form of additional dependencies between the stereotypes. For example, the operation overriding convention (see previous section) can be expressed as a dependency from ConcrObserver’s “update” to Observer’s “update”, with a constraint that enforces the matching of the signatures of the operations bound to these roles.
The entire profile extracted from the Observer pattern becomes a fairly complex class diagram, and we will not present it here. However, it should be clear that we can express all the structural requirements of the pattern, as discussed in the previous section.

An instance of the pattern as it would appear in a model is presented in Figure 11. The use of the conventional pattern instance symbol in UML (collaboration ellipsis) is not necessary here since the stereotypes indicate both the existence of a pattern instance and the roles different parts in the model are playing in the pattern. However, in a real model where stereotypes are used for other purposes as well and where there may be several instances of a pattern, a separate symbol marking a pattern instance is needed.

Fig. 11. An instance of the Observer pattern

4. From Profiles to Patterns

The transformation process described in the previous section can be reversed, allowing us to produce a pattern model from every profile constructed according to our principles. According to these principles, stereotypes can be introduced for any metaclass, and structural constraints can be expressed by specializing existing meta-associations. The idea is simple: if we interpret every stereotype class in the profile as an instance of the corresponding metaclass, and consider the associations between the stereotype classes as links between these instances, we get a valid instance of the metamodel. This follows from the fact that the associations between stereotype classes are only specializations of the meta-associations between the corresponding metaclasses. Thus, there must be a visual presentation of this model instance. If we put this visual presentation inside a pattern diagram rectangle, we have the proper patterned description of the profile. Note that the transformation rules of Figure 9 must be reversed to generate the “any” instances according to the use of the {strict} constraint in the profile. For example, the result of transforming the profile in Figure 4 into a pattern is presented in Figure 12.

Again, the handling of multiplicities appearing in the profile specification is less straightforward. However, assuming that multiplicities are used only in a restricted manner, which guarantees that the multiplicities can be transformed into cardinalities of certain roles in the profile, multiplicities can be allowed in a profile without hampering the transformation process. Similarly, the use of special dependencies with constraints must be limited to some standard, known set.
It should be emphasized that this transformation produces only a model fragment, and we assume that the <<pattern>> stereotype gives this fragment the correct interpretation of a pattern (to be used, for example, by a tool for instantiating the pattern). Similarly, the reverse transformation from patterns to profiles produces only a profile which can be used as a basis for checking certain structural constraints implied by a pattern, but not for locating instances of patterns. The notion of a pattern instance cannot be captured by the simple profile specification mechanism we use. For example, the model of figure 13 would be accepted as a legal model according to the Observer profile, although there are no valid instances of pattern Observer in the model (we have omitted the operations).

Fig. 12. Patternized profile

Fig. 13. An accepted model without legal pattern instances
5. Related Work

Patterns and profiles have been previously discussed in the same context in [12], where a UML profile is derived to support the presentation of design patterns. However, their approach is different from ours in the sense that they propose a general profile for all possible patterns, while we consider a single pattern as a profile. Their purpose is not to unify patterns and profiles, but rather to show what kind of a profile is needed to support patterns in UML based software development in general. Similarly, Fontoura and Lucena [4] propose UML extensions (mostly tagged values and constraints) to support the presentation of flexibility points of design patterns. They also present a technique to specify the supposed instantiation process for such a pattern as an activity diagram. A general profile for specifying the extension interface of an application framework has been presented in [5], exploiting tagged values.

Perhaps closest to this work is the technique described in [13], where special notational conventions are used to express certain structural requirements for a UML model. Using these conventions, the architect can specify structural rules as a profile. Integrated with Rational Rose, the technique allows the checking of a design model against the architectural rules of, say, platform or product-line. The rules in a profile are given in the style of examples of the required structure. In some cases such examples come close to a pattern-like representation, although the forms allowed in [13] are more restricted and do not rely on the pattern concept. From this viewpoint, our work could be seen as a generalization and systematic derivation of a profile description language.

6. Discussion

From the methodological viewpoint, the unification of patterns and profiles opens up new possibilities. In particular, the architect can conveniently describe a profile as a collection of patterns, to be followed by the designers. Indeed, the technique described here allows, at least in principle, the interpretation of any model fragment as a profile, taken from any diagram type (ignoring the need for different levels of strictness of the rules). For example, a sequence diagram or a state diagram can be interpreted as a behavioral profile. We anticipate that this kind of “profilization” of models can be very useful especially in the context of product-lines: the architect can give a prototypical model of an application based on the product-line, and turn this into a profile to be followed by the application developers. Presenting a profile as a set of patterns is attractive from the comprehensibility viewpoint as well: an example-like pattern is much easier to understand than the fairly cryptic metamodel extension form.

From the viewpoint of UML tools, our work allows the integration of the tool support for patterns and profiles, which has been the primary motivation for this work. On one hand, we have developed an environment for pattern-driven software development, where the patterns guide a system developer and generate software artifacts ([8], [9]). On the other hand, our team has experimented with a tool set for specifying structural profiles for the purpose of model checking [13]. We anticipate that the results presented here allow us to combine the benefits of these different tools: a single tool environment, based on a unified pattern-profile concept, can both assist in the production of models according to predefined rules, and in the checking of the models against profiles. Furthermore, this approach allows UML itself to be used in a natural way as the description language of pattern-profiles.

We believe our existing tools reflect the purposes of patterns and profiles according to their original intentions. Patterns have been the means to describe individual design
solutions, whereas profiles have been understood as domain specific refinements of the modeling language. Being able to express patterns and profiles in a unified fashion is the first step in our attempt to integrate the pattern and profile features of our tools for a true unified architecting environment.

Acknowledgements. This work is supported financially by the Academy of Finland (projects 51005 and 51528) and by the National Technology Agency of Finland (project 40226/04). We wish to thank Mika Siikarla for suggesting the use of the OCL template in profiles, and professor Tarja Systä for many useful discussions.

References

An Approach for an UML Profile for Software Development Process Modeling *

Ralf Buschermöhlê and Wilhelm Hasselbring

1 Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS) e.V.
Department Safety Critical Systems
Escherweg 2
D-26121 Oldenburg, Germany
Email: buschermoehele@offis.de

2 University of Oldenburg
Software Engineering Group
Department of Computing Science
D-26111 Oldenburg, Germany
Email: hasselbring@informatik.uni-oldenburg.de

Abstract. The status of our current work on an UML profile to express software engineering process models is presented. We will discuss our requirements on such a profile and introduce identified syntactical and semantical implications. Concluding a first application scenario is presented.

1 Introduction

Nowadays various development process models exist for efficient development of software. They range from agile processes, like eXtreme Programming (XP [5]) as a set of “best practices”, customizable frameworks such as the Rational Unified Process (RUP [21]) towards very detailed process descriptions such as the V-Model [17]. They all have in common that the process specifications are expressed informally. This often leads to different interpretations which is obviously a problem for (automatic) process execution. The informal description of development processes leads to the problem that it is not well understood what should be coordinated/expressed in a software development process and what not: “To which extent should a software development process be specified?”

There exist a lot of formal software process modeling languages which allow to avoid ambiguities when interpreting the process specifications [18, 19, 15]. Unfortunately none of them achieved the status of a standard notation for software process modeling. All of these approaches define their own syntax and semantics which implies e.g., proprietary data formats. Furthermore all of them are limited to express a fixed set of certain aspects of a process software model (e.g., time, data, concurrency) and it is not possible to extend them easily with new modeling constructs e.g., to cover new research topics like “knowledge management” or different semantics (e.g., finite automata, Petri net variants).

On the other hand the Unified Modeling Language (UML [37]) of the Object Modeling Group (OMG [1]) achieved the status of an industrial standard for modeling object oriented systems which resulted in considerable tool support. Even more, UML provides easy to use extension mechanisms (stereotypes, tagged values, constraints) within so

* This research was supported by the German Department of Education and Research (BMBF), research project ’ViSEK’, under grant 01ISA02G.
called "UML profiles". Unfortunately the UML does not provide formal semantics. There are different approaches published trying to solve this issue (compare Section 2). But they all have in common that they focus only some aspects of the UML diagram conglomerate.

For these reasons we present an UML-based approach to support software development process modeling and execution. The remainder of the paper is structured as follows. In Section 2 we briefly introduce some informal basic definitions and motivate the approach. In section 3 we give a brief overview of the accomplished work. Section 6 concludes and elaborates on further research.

2 Fundamentals

In this paper we discuss the "software dimension" of software development processes, this includes all software related elements that are used in the process, like artifacts and tools. We aim at automating as much of these software elements as possible. We start the discussion with some requirements sketching the "big picture". Then we discuss briefly implications of these definitions with respect to modeling static and dynamic concerns inside an UML profile. Requirements of software development process modeling are:

1. A software development process consists of (disjunctive) sets of activities, actors, products and tools ((A)APT).
2. Each AAPT can be ready or not ready and describes for example the availability of an actor. Even more a resource can be several "times" ready meaning that several products are available.
3. Each activity has a set of inputs and outputs that refer to a subset of defined AAPT.
4. If all referred AAPTs of an input are ready then the input is enabled.
5. An activity is (possibly) executed if at least one input is enabled.
6. A step is subset of (possibly executed) activities that is executed concurrently. Especially the concurrent execution of two equal activities is possible e.g., regarding a "coding" activity where more then two persons code at the same time.
7. If an activity is executed the corresponding input is disabled and it’s referred AAPTs are set to not ready.
8. If an activity completes at least one output is enabled.
9. If an output is enabled all referred AAPT are ready.
10. Each APT has a set of attributes to store data and states (e.g., actor name, product state).
11. The APTs may have relations among each other (e.g., product a is related to exactly 3 subproducts).
12. Tools are able to change states of products within a defined set of functions that map a set of products in certain states into a set of products in certain states.
13. Actors can be persons interacting with tools or just tools.
14. Actors interact with tools in that way that certain functions are executed.
15. An application of these functions is referred as activity execution which can consume resources.
16. Resources can be time, cost, space etc.
If one tries an initial (syntactical) mapping of the identified requirements towards UML diagrams it is obvious that we need to be able to model static and dynamic aspects of a process. The kind of static aspects we need to express are covered in UML only by class diagrams - so there is no design choice. But there are a lot of diagrams dealing with behavioral aspects like state charts, activity charts and message sequence charts. If we assume, that the static instances of actors, products and tools can be described by class diagrams then activity charts offer an adequate syntax to meet the required inputs and outputs (requirement 3). Regarding possible semantics the main characteristic of the process dynamics we sketched so far is "resource driven". Which means, if all required resources are available or ready the activity is executed. Including all UML specifications until version 1.5 [37] the OMG proposes to derive the behavior of activity machines from state machines. This would not be consistent with our requirements e.g., it is not possible to have multiple active states in one state chart. Fortunately, in the actual UML 2 proposals [26] the OMG is changing the behavioral of activity machines towards a Petri net-based semantics which match our identified requirements. A Petri net PN is a tuple

\( (P,T,F,B) \)

- a finite, ordered set \( P = \{ p_1, \ldots, p_n \} \) of places,
- a finite, ordered set \( T = \{ t_1, \ldots, t_m \} \) of transitions,
- \( P \cap T = \emptyset \),
- \( |P| \times |T| \to \text{forward - matrix} F \) over \( \mathbb{N} \), and
- \( |P| \times |T| \to \text{backward - matrix} B \) over \( \mathbb{N} \)
- \( F : P \times T \cup T \times P \to \mathbb{N} \) the edge function, defined as \( \forall x, y \in P \cup T : \)

\[
F(x,y) = \begin{cases} 
F_{i,j}, & \text{if } x = p_i \land y = t_j \\
B_{i,j}, & \text{if } x = t_j \land y = p_i 
\end{cases}
\]

- The State space of the Petri net is \( \mathbb{N}^P \).
- A mapping \( s : P \to \mathbb{N} \) is called state/marking of the net.
- There exist exactly one initial state \( s_0 \).
- We call a transition \( t \) of \( N \) s-enabled, if \( s \geq F(t)(\forall p \in P : s(p) \geq F_{p,t} = F(p,t)) \).
- \( t \) fires from \( s \) to \( s' \), if \( s' = s - F(t) + B(t) \).

Figure 1 shows a simple example of a Petri net-based execution frame for activity charts. In sub-figure (a) the activity chart is presented and in sub-figure (b) the corresponding Petri net. Note that is is not necessary that "Activity_{i+1}" is ever reached, because \( T_0 \) or \( T_1 \) can also fire twice in standard Petri net token game semantics. Petri nets are widely accepted in the area of business process models regarding the execution of so called "workflow specifications" in so called "workflow management systems" [3, 31]. But workflows differ from our point of view from our software process models - which reflects also in the Petri net token game semantics that is used to describe the behavior of workflows.

One characteristic of Petri nets is that they generally describe closed systems. This assumption is appropriate if the process model should "only" be simulated but this is not a realistic assumption if we want to accompany a "real" software development processes where we possibly have to deal with variations over time regarding the resources that can be used. So we need to support semantics that allows to specify so called "reactive systems". Furthermore as one of their main characteristic workflow management systems do not allow to modify the controlled artifacts. But exactly this is an important feature for automating software development processes.

There exist a lot of approaches to express Software Engineering Process Models in UML and UML profiles (e.g., [30, 34, 35]). Most of them have an informal semantics.
and thus they are outside of our scope — since we aim at allowing (computer supported) software process execution. Regarding the formal semantics one could classify the approaches regarding the level of UML coverage. Many people have defined the semantics of single UML diagrams — e.g., [22, 7] etc., on state machines, [12] on collaboration diagrams, [13, 16] etc. on class diagrams, [28] on use cases, [6] on activity diagrams — or just to give formal foundations for action language (e.g., [25, 2]). This restriction on a single diagram is problematic because the main advantage of UML is the possibility to use different diagrams during the model building process to describe a system. Most of the diagrams are related to each other and thus the formal semantics should have ”interfaces” to connect to other approaches in order to complete the picture.

One example of such interfaces in our approach concerns the tool dimension of a software development process. All software development processes have in common that tools are required to support their activities [32, 14, 38]. This is due to the fact that process models usually specify the activities that should be accomplished and the tools determine the activities that can be accomplished — with various levels of automation. Similar relationships between artifacts and process models resp. tools exist. The relevance of these mutual dependencies is proportional related to the amount of activities for producing artifacts that can be supported by tools. Only then, tools are able to serve as bridges between the process models and the actual processes. The amount of possible automation is steadily increasing — especially in the domain of system development — since only tools are able to support the efficient development of the systems that have a continuously increasing complexity [11]. The relationship between process models and tools is relevant in both directions. On one hand, process models have an impact on the tools. On the other hand, the tools have an impact on the process models. The main concern for this direction is the fact that software development processes are similar to software or at least several aspects of them are expressible in software. Therefore, we have to deal with the question how the software environment that is used in a development process can be integrated into
a process modeling language. In the next section we are going to introduce some of our design choices and results so far.

3 Work status

First we developed a mapping regarding the syntactical constraints of an established development process model in order to see whether all required elements of the process specification could be expressed. Then we expressed this process model with the chosen UML subset. After this we extended these models with additional activities and products concerning domain specific standards in the area of safety critical systems (like ARP4754 [33] in the area of avionics) and also introduced model-based development extensions (like Life Sequence Charts [9]) to be able to verify systems very early in the process. For this example mapping we used the V-Model for its widespread use in the domain of safety critical systems, and its property of being both very detailed and general. Furthermore this process model is mandatory for software suppliers working for German government organizations. Activities in the V-Model context

"are work-steps in the IT development process; its results and execution can be described exactly. Activities may consist of a set of sub-activities as long as each of these sub-activities results in defined interim results." ([17], GD250-2EINF, p.5)

In each activity objects are processed, called products. And all products that have to be developed are described (method independently) in detail in the specification which serves also as contract between customer and contractor.

The V-model comprises the four submodules project management (PM), configuration management (CM), quality assurance (QA), and system development (SD). We concentrate on the most elaborated part SD. Instead of insisting on particular languages for products, the V-model in detail specifies what a product shall describe and recommends specific methods, i.e. languages or notations such as flow-charts, or even gives general product templates to be used in particular situations. A product template, for example, specifies that each requirement must have a unique identification number. Furthermore on the method level there are "method interfaces" specified that explain how the methods relate to each other. In order to illustrate the mapping of the V-Model specification elements towards the UML profile elements we will sketch some examples in the next two subsections.

3.1 Activity related specification elements in the V-Model

There are two main specification elements that cover the execution of activities and methods in the V-Model, e.g. "V-Model activity diagrams" and product flows. V-Model activity diagrams are used to sketch a grain granular picture of the relationships between a set of activities and products (example see figure 2). Activities are depicted in rectangles and products in ellipses. Control- and data flows are displayed by arrows, whereby data flows have at least one product as source. These "pictures" have not much in common with the specifications on deeper levels because the are only data flows specified. As already mentioned the lower specification elements provide a more detailed picture of the development activities. Each activity has a so called "product flow" that describes all input
Fig. 2. Example Activity SD 2: System Design (V-Model Spec.)

and output products. Figure 3 describes all products involved in activity SD 2.5 "Interface Description". For each (sub-) product (column 3) referred to by the activity to be described the state at the beginning (column 2) and that at the end of the activity (column 5) is entered. If the activity does not influence the state of the product or should no such state of the product exist, then this is marked in the table by a dash. The input products of an activity are identified in such a manner that the columns "from Activity" and "from State" are filled in and the columns "to Activity" and "to State" are marked by a dash. For output products the "From"-entries are not applicable. Only the columns "to Activity" and "to State" are filled in. In the cases where a product has both "from" and "to" entries it is modified in the corresponding activity. All output products of an activity whose end states are "b. proc." should have "planned" as beginning states according to the model. In order to be able to distinguish better visually between input and output products, the beginning state has been substituted by a "—". This should be interpreted as "planned" in these cases. Furthermore, it is noted for each (sub-) product, from which activity the product results (column 1) and to which activity the product will be passed (column 4). If there are neither "from" nor "to" activities for a (sub-) product this is illustrated in the table by a dash. If sub-products of a product are created in different (sub-) activities (see, e. g., activity QA 2.2 "Definition of Assessment Environment"), it will become necessary to assemble the product by integrating the sub-products. This is realized in the activity where the last generated sub-product of a product is created. In the product flow this is represented by referring to following main activities in column "to Activity". For products
that are not updated, the state in "to State" is "submitted".

The example shown in Figure 3 means that "User Requirements" and "System Architecture" come from activities SD1 and SD2.4, and represent input products. Both products have state "accepted". Product "Interface Description" is newly created. The product leaves the activity having the state "proc" and is input product for activities SD2.6, SD3, SD4-SW, CM4.3. Figure 4 shows an aggregated activity diagram of figure 2 and the relevant part of figure 3. In the activity diagrams all resources (products, actors, tools) are objects. Furthermore in the diagram of figure 4 the objects have a state that corresponds to the "product flow" of the V-Model. The next activity level that is specified in the V-Model are the methods. There are a lot of information that describe for example what to do in each method, limits and recommendations during the method application or sketch interfaces among the methods. But the assignment of methods to activities is done by allocation a set of methods to each activity. Figure 5 shows an assignment of the methods "Class and Object Modeling" (COM), "SubSystem Modeling" (SSM), "Formal Specification" (FS), "Design VERification" (DVER), "Analysis of Covered Channels" (ACC), "State Machine Modeling" (STMO) and "InterAction Modeling" (IAM) to activity SD2.5 "Interface Description". We further assumed that methods are just activities without a specified activity flow because this would exceed the limits of a process model description (for example the V-Model specifies over 60 methods). Nevertheless it
would be useful to have such descriptions - especially if most parts are inherently dy-
namic between most process runs. Figure 6 depicts a possible flow of methods. First the

input products of the super activity (SE2.5) are instantiated, namely the objects "System
Architecture" (SA) and UR (User Requirements) according to the product flow table of
the activity. Then these products are split into the required sub-products that should be
modified by method application, namely in figure 6 "System Architecture.Use Cases"
(SA.UC), "System Architecture.Class and Object Model" (SA.COM) in state "s1" and
"System Architecture.State Modeling" (SA.CSTMO) in state "s3". The methods "Class
and Object Modeling" and "State Modeling" should be executed sequentially infinitely
often till both produced the objects "SA.COM" in state "s2" and "SA.CSTMO" in state
"s4". The changed states denote in this diagram an object change. What changed exactly
is specified somewhere else. Figure 7 shows the corresponding underlying Petri net se-
mantics. Now we are able to use activity diagrams for software process modeling. But we
did not show yet how static aspects (data) can be expressed. Even more we assumed that
objects are "created" - but we did not specify how. According to our defined requirements
we would assume that tools exist that are able to create, modify und (possibly) destruct
the products.

3.2 Product related specification elements in the V-Model

There are several product related specification elements in the V-Model. This includes
for example general product information (like: name and identification of the document,
version, person in charge), product information (for example for user requirements) and
activity-related product information (like: user requirements that describe the actual status of an existing old system, that should be (partly) incorporated into the new one in activity SD1.1 "Recording Actual Status and Analysis"). Even more relationships between methods are specified, for example

"Interface Design Verification - Formal Specification
DVER requires a formally specified detailed specification for to be verified and formally specified starting specification. These specifications should be written in the same specification language."

We mapped these specifications into UML class models. Within their syntactical boundaries we were able to express

- Product relationships with multiplicities that denote that a product has a relationship to other products, for example each state chart has exactly one class diagram.
- Product instances that describe inheritable product attributes, for example each products inherits all general product requirements.
- Product hierarchies describe product affiliations, for example that product "System Architecture" consists of class diagrams, state charts and a design verification proof.

Figure 8 depicts some method results of activity SE2.5. The "Interface Description" product has the two sub-products "Class and Object Model" and "State Model". The "C" at the beginning of the class names show that we talk about "Classes" - thus product templates in the V-Model context. Furthermore each of the sub-products of the class "Interface Description" inherits the attributes "InterfaceID" of type "Unique ID" (UID) and a string that informs about the purpose of the interface. Even more relation "R3" ensures that each class has exactly one state chart.
Comprising the above we were able to express specification elements that were syntactically clear separable e.g., product flows, method assignments and method interfaces inside a UML profile that consists syntactically mainly of activity machines and class diagrams. We did not express large text elements in "pure" natural language directly, since the possible interpretation range would be impractically wide. In this case we referred in the diagrams to the unique specification ID. Nevertheless we assume that it is possible to express these elements also inside the chosen UML set. We introduced our models on the last user conference of the V-model, called "ANSSTAND" in October 2003 [4].

4 Application Scenario: Process Model Monitoring

Unfortunately the diagrams provide a very grain granular picture of the development activities that have to be done because they are not complete. Regarding the main activities the flow was incomplete and unclear product flow as well as control flow. Regarding the activity level (every dynamic aspect above the method level) it was for example unclear what products should be locked and it is obvious that the QS states are not enough to express all (relevant) product changes. Regarding the method level the V-Model authors specified just a set of methods but did not how these should be executed. In several discussions during the ANSSTAND user forum and in several other meetings with industrial partners it turned out that exactly these were the major problems when the V-Model is applied. Even more when using process models a lot of experience is needed to ensure that a process is executed in the "right way" due to several from project to project changing context factors, like process targets (time-to-market, budget, quality) resources (people, software) and evolving methods and techniques. Nevertheless we believe that it is possible to "sharpen" these pictures of a process with appropriate tool support. Figure 9 shows the "big picture" regarding a process model monitoring. Process model monitoring means tracking a certain set of process information and comparing them with an actual process model to find out how "good" the development works. There exist already some approaches in this area but their problem is that the tracking is mostly manual which suffers from several drawbacks, like:

- Incompleteness because often only a certain detail level can be tracked.
- Cost and time intensiveness because extra personal is needed to track the development activities and their results.
– Disturbance of the "normal" development activities.
– Faults in the mapping towards the chosen PML.
– Inconsistencies of several development tracks, especially when different "trackers" are involved.
– Snapshot character of the captures because the tracking is done once or twice and often even not the whole life cycle of a product.

For these reasons we try to develop techniques that are able to sketch development activities more automatically through a "tool’s perspective" [8]. The first step in this di-

![Diagram](image)

**Fig. 9. Process Monitoring**

rection is in figure 9 depicted. The arrows in this figure denote information flows. As already mentioned the UML diagrams of the V-Model are not complete. Nevertheless these diagrams provide a (first) base to structure the development. For this reason we use these information to inform a process executer via the "Mediator" about possible process steps that should be executed (A). Afterwards the process executer uses the "Mediator" to inform about the artifacts that were developed (B). All the modified or created artifacts are analyzed by a static analysis based on the JavaCC parser [20] to see what changes in the artifacts (for example the number of methods in a Java program). Furthermore we mapped the grammar files also in UML class diagrams that describe the static structure of the artifacts in the process model context. Therefore we are able to draw (detailed) versions of the V-Model templates according to the modified and created artifacts (D). Since we would like to provide a "post mortem analysis" of the executed process we also store the artifacts on a CVS/DB server (C). If enough of these process runs are captured they have an influence of the templates (E).

### 5 Formal Kernel Semantics

In the first step to formalize the chosen UML subset we defined a so called "kernel semantics" that can deal with every element of the syntax but provides only the necessary semantics on a low level. This is done because the developed semantics is much easier to handle if it is necessary to add additional constructs later on and we can already integrate necessary "interfaces" to other semantics as needed. The two bases for these semantics
are the activity diagram semantics introduced in the thesis of Erik Eshuis [31] and the kernel language of the state machine semantics developed by Damm et.al [10]. The activity machine semantics was chosen because Eshuis developed so called "reactive" Petri nets that are able to deal with events. But his semantics do not include several aspects activity machines offer like an action language, object flows or triggered operation calls. The state machine semantics was chosen to have a base for a kernel semantics that can deal with every aspect of state machines in order to complete the "big picture". This section will only give a short overview on the developed semantics in order to sketch ones of the differences between activity machine and state machine semantics we had to deal with. A krtUML model \( M = (T, F, \text{Sig}, <, C, \text{c}_\text{root}, A) \) consists of the following:

- \( T \supseteq \{\text{void}, \mathbb{B}, \mathbb{N}\} \): A set of basic types comprising at least booleans and natural numbers.
- \( F \): A set of typed predefined primitive functions.
- \( \text{Sig} \): A finite set of signals. Every instance of a signal is called signal event, or event for brevity.
- \( < \subseteq \text{Sig} \times \text{Sig} \): A generalization relation on signals, i.e. the transitive closure \( <^+ \) is irreflexive, where \( ev_1 < ev_2 \) denotes that \( ev_2 \) is a generalization of \( ev_1 \). In the following, we use \( \leq \) to denote the reflexive transitive closure of \(<\).
- \( C \): A finite, non-empty set of classes. A class \( c = (c.\text{isActive}, c.\text{attr}, c.\text{ops}, c.\text{sm}, c.\text{act}) \)

  consists of:
  - \( c.\text{isActive} \): A predicate. Class \( c \in C \) is called active iff \( c.\text{isActive} = \text{true} \).
  - \( c.\text{attr} \): A finite set of typed attributes, which may not be of type \( \text{void} \).
  - \( c.\text{ops} \): A finite set of typed triggered operations.
  - \( c.\text{sm} \): A c-state machine in terms of c-actions over c-expressions.
  - \( c.\text{act} \): A c-activity machine in terms of c-actions over c-expressions.
- \( c.\text{root} \in C \): The class of the root object (serving to specify system initialization).
- \( A \subseteq C \): A subset of active classes called actors and used to denote external objects (part of the environment).

Each definition element (e.g., Class, Action, Expression, Guard) is typed consisting according to the defined basic types. An c-activity machine for a class \( c \in T_C \) is a tuple \( c.\text{act} = (c.Q, c.q_0, c.Q_x, c.\text{Tr}) \), where:

- \( c.Q \) is a finite set of activities.
- \( c.q_0 \in c.Q \) is the initial activity.
- \( c.Q_x \subseteq c.Q \) is a set of termination activities with \( c.q_0 \notin c.Q_x \).
- \( c.\text{Tr} \subseteq \{S | S \subseteq c.Q\} \times \{\{\gamma \mid \gamma \text{ is a c-guard or c-action}\}\} \times \{T | T \subseteq c.Q\} \) is the transition relation and \( \forall \text{tr} = (S, \gamma, T) \in c.\text{Tr} \):
  - \( c.q_0 \in S \Rightarrow S = \{c.q_0\} \land \gamma = \text{"create}_c\text{"} \)
  - \( c.Q_x \cap T \neq 0 \Rightarrow T \subseteq c.Q \).
- Class \( c \in C \) is called reactive if there is a transition \( (S, \gamma, T) \in c.\text{tr} \) such that \( c.q_0 \notin S \) and \( \gamma \) is in the form \( ev[\text{expr}] \) or \( op[\text{expr}] \) for some \( ev \in \text{Sig} \) or \( op \in c.\text{ops} \setminus \{\text{create}_c\} \).
We manage objects in so called object configurations that we assume to exist for each object. Such object configurations store the status (e.g., dormant, idle, executing, suspended, dying, dead), the attribute configuration (the values of all attributes), the configuration (in a state machine the active state, in an activity machine similar to a marking of a Petri net) and the event queue of an object. The system configuration is a set of all object configurations.

We use so called pending request tables to store triggered operation calls. Each triggered operation call consists of the destination of the call, the status of the operation (unused, pending, busy, completed), the result of the operation and the parameters of the operation. One of the main differences between objects of a state machine and objects of an activity machine we made was that the first ones can be grouped inside a "component" where only one of the objects is active in a certain point in time. The rest of the objects is not executed (for example dormant or dead) or suspended due to pending operation calls to other objects. In contrast activity machine objects are always active because of their resource driven token switching. This has for example an impact on the destination attribute of a pending request table. If the destination of a call is a state machine object then the destination field contains a reference to one object of the component. Otherwise the destination field contains a transition of an object’s activity machine. The semantics of the krtUML was defined in terms of a "symbolic transition system", proposed in [23] under the name Synchronous Transition System. In such a system all variables are mapping to values of their domains in so called "snapshots".

A symbolic transition system (STS) \( S = (V, \Theta, \rho) \) consists of \( V \), a finite set of typed system variables, \( \Theta \), a first-order predicate over variables in \( V \) characterizing the initial states, and \( \rho \), a transition predicate, that is a first-order predicate over \( V, V' \), referring to both primed and unprimed versions of the system variables (their current and next states). An STS induces a transition system on the set of interpretations of its variables as follows. Let \( S = (V, \Theta, \rho) \) be an STS and \( T \) the set of types of variables in \( V \). Let \( D_\tau \) be a semantic domain for each \( \tau \in T \).

- A snapshot \( s : V \to \bigcup_{\tau \in T} D_\tau \) of \( S \) is a type-consistent interpretation of \( V \), assigning to each variable \( v \in V \) a value \( s(v) \) over its domain. \( \Sigma \) denotes the set of snapshots of \( S \).

- A snapshot \( s \in \Sigma \) inductively defines the value \( [\text{expr}](s) \) for first-order predicates ‘expr’ over \( V \) and the value \( [\text{expr}](s, s') \) for first-order predicates ‘expr’ over \( V, V' \), where \( s \) provides the interpretation of unprimed and \( s' \) the interpretation of primed variables in ‘expr’.

- A snapshot \( s \in \Sigma \) is called initial, iff \( [\Theta](s) = \text{true} \).

- Let \( s, s' \in \Sigma \) be snapshots of \( S \). Snapshot \( s' \) is called \( S \)-successor of \( s \), iff \( [\rho](s, s') = \text{true} \).

- A computation, or run, of \( S \) is an infinite sequence of snapshots \( r = s_0 s_1 s_2 \ldots \), satisfying the following requirements:
  - Initiation: \( s_0 \) is initial.
    - for each \( j \in \mathbb{N}_0 \).
  - The set of all computations of \( S \) is denoted as \( \text{runs}(S) \).
Thus we elaborated in this semantics the way the snapshots can evolve, defining for each of the possible cases a transition predicate. Finally, we defined the predicate characterizing initial snapshots and collect all pieces of the transition relation into a full predicative definition of the transition relation, leading to a definition of the symbolic transition system associated with the $krtUML$ model.

5.1 Example interface between state and activity machines for process monitoring

Objects are needed to express various process related information, like all kinds of artifacts, persons and software. Objects are created and live until they are destroyed. In general they should have a system-wide scope, but it must be possible to put them into local scopes (e.g., an activity). Furthermore they have additional attributes and methods for example to support product locks for write access. Events are needed because it is not always possible to distinguish two product according to their state (for example imagine refactoring activities) or to model timers (and thus timing events). In contrast to objects events live exactly one step after the are send (no fixpoint semantics) and they are always related to a certain element (object, transition).

Central for the whole discussion of software process modeling are objects and their way they are treated inside activity diagrams. As already depicted in figure 1 the first approach is to sketch the artifacts an activity ”consumes” and ”produces”. This picture is complete for tracking purposes as long as we don’t want to monitor software development processes. If we want to monitor a software development process than we must be able to react on a certain set of objects that is developed during the execution of an activity and integrate these objects into the picture. But this set of objects can’t be specified initially before the process is executed because we don’t know how many objects we will have at a certain point of time. We additionally required that tools are able modify this object set. But tools don’t ”behave” on base of a Petri net based execution semantics. They react on a state chart-based semantics. Thus we need to discuss the possible combination of activity-charts and state charts in relation to what these diagrams should be able to exchange. As already mentioned we need objects as permanent ”things” and events as temporally restricted ones to a single step. Regarding an object modeling we made these design choices:

1. There exist object places for all objects $o_1, \ldots, o_n \times \{\text{lock, unlocked}\} \times \{a_1, \ldots, a_m, \text{no scope}\}$ all activities.
2. The object places are connected with the activities that produce and consume the objects.
3. There exist two additional places for each object as object token sources and drains.
4. Each time a creation of object $o_i$ is executed a token is put onto it’s object token source.
5. Each time a destruction of object $o_i$ is executed a token is removed from one object token place (that is designated for token removal).
6. Tokens are inside a scope of an activity if the tokens are moved to the designated places of the activities.

Figure 10 shows a simple Petri net invoking a tool a. This figure bases one the activity diagram of figure 1, whereby on ”Activity,i+1” a ”tool (a)” in invoked — this can be expressed in the activity diagram by an object ”a” with stereotype ”tool”.

One difference compared to figure 1 is that the object state space is a subset of the Cartesian product we defined in our design choices. Furthermore figure 10 depicts token
Fig. 10. Simple Petri net semantics including tool sources and drains as connection between activity and state machines, illustrated as transition without source place or without destination place. If it is not possible to determine object changes according to their object states then events should be used to inform the activity-machine about ongoing process. Regarding events we made these design choices:

1. There exist event places for all events that can be received.
2. An event starts to "live" one step after it it sends (no fix-point semantics).
3. An event lives exactly one step.

Figure 11 shows a Petri net distributing an event. After informing the event source that the event occurred the token source puts five tokens onto the event distribution place (ed). Then all event consumers (ec1, ..., ec5) can fire.

When distributing an event it is necessary to remove the unused event tokens after one step from the distribution place. This is done by transition t6 in the example. Notice that we have to make three assumptions regarding the firing of transitions compared to place/transition Petri nets. The first is that we have a maximality constraint that says that in each step the set of fired transitions include as many enabled ones as possible. The second is that we have two steps when choosing the set of enabled transitions that will fire. The first one includes all transitions without that ones that have "sink" destinations. The second one includes the other transitions. The third is that event consuming transitions are only allowed to fire once in one step.

To model the diagrams we used Together 6.1 [36] - mainly because of three reasons. It is the only UML tool we know that allows to express activity machines in such an elaborated way (for example Rational Rose [29] v2003 does not support object flows). The second reason was that Together offers a wide range of possible XMI [39] variants in order to exchange models. Unfortunately the exports of version 6.1 were not compliant to
the OMG meta-models but it was possible to develop a mapping. With these changes we were able to parse the models into an OFFIS [27] development of a MOF [24] repository.

In order to summarize the first results so far, we developed a syntactical mapping of a process model towards a snapshot of the UML language, the Kernel semantics for the execution of activity machines in relation to state machines and an architecture including a first prototype implementation of a framework to track the execution software development processes.

6 Conclusion and Further Research

The status of our work on defining an approach for a UML profile for software development process modeling was presented. We started with a motivation for such a profile and defined general requirements. Then we briefly discussed why the previous work on UML formalization do not match and explained our approach shortly.

Our next research activities focus on the definition of the precompilation phase and the implementation of the software based process monitoring framework. The precompilation phase describes the unfolding of all used diagram elements into the specified kernel model language. Afterwards this language has to be extended in a direction to be able to express software development process templates. In such a template we have to deal for example with the question what are universal and what existential elements. At the same time we plan to extend our prototype implementation towards a stable and easy to use software system in order to get more "real" world process descriptions. After extending the capabilities of the kernel language with respect to the specification of process templates we have to deal with the question how the framework is adoptable to this requirement. We are going to integrate the "Mediator" component more and more directly into the development tools so that we are directly informed when objects are created, modified or destroyed.

References


UML Profile for Analysis and Design of Jakarta Struts Framework Based Web Applications

Piotr Karwaczyński¹ and Łukasz Maciejewski²

¹ Division of Computer Science, Wrocław University of Technology, Poland
piotr.karwaczynski@pwr.wroc.pl

² Institute of Materials Science and Applied Mechanics, Wrocław University of Technology, Poland,
lukasz.maciejewski@pwr.wroc.pl

Abstract. Jakarta Struts Framework is a mature open source solution for the development of Java-based web applications. In this paper the authors propose UML profile supporting such development process. The profile is adapted to five models describing different stages and viewpoints of analysis and design of Struts-based applications. As the profile and the range of chosen models proved their usefulness in real life scenarios, the fragments of commercial web application model are also presented.

1 Introduction

A combination of JSP scripting language, Java language and Jakarta Struts Framework has become one of the most popular solutions for the web applications development. JSP and Java are being developed by Sun Microsystems whereas open source Struts Framework is a part of the Jakarta Project led by Apache Software Foundation.

Tools mentioned above are mature and widely used. They speed up the process of the web application development significantly. Nevertheless, they do not offer any accompanying solutions for analysis and design phases. The authors, basing on their experience, point out two major deficiencies:

– lack of the suitable notation for the modeling of web applications,
– lack of tools for the visual modeling of web applications.

The general notation focused on the web applications development process has been introduced by Connalen [1]. However, it does not express all the abstractions introduced by Struts Framework. Thus the authors propose a specialized notation ready to use with Struts Framework. It is developed as an UML profile [2] dedicated for the modeling of Struts-based web applications. Its main objective is to raise the quality of the documentation of Struts-based web projects without introducing a significant overhead usually related to the documentation process.

From among tools supporting the Struts-based web development it is worth to mention Struts Studio by Exadel and Eclipse (an open source IDE) with a set of appropriate plugins. This tools enable a developer to:

– generate a basic web application structure (directories and sample files),
– generate a basic web application configuration files,
– edit configuration files using their graphical visualization,
– partially visualize control flows in the scope of a web application.

Unfortunately such a graphical visualization is merely a part of an application model and does not cover all the web developer need in analysis and design. The authors assume that the proposed set of models may be incorporated into existing development tools (e.g. Eclipse) as a specialized plug-in and utilized effectively.
2 Jakarta Struts Overview

Jakarta Struts Framework, likewise Tomcat, Ant or Velocity, is a part of the Jakarta Project held by Apache Software Foundation. Since June 2001 Struts has been available as an open source under the Apache Software License. It means that Struts may be utilized to develop web applications of any kind free of charge.

Jakarta Struts Framework bases on an architecture model commonly known as the Model 2. It has been described in the Servlet/JSP Specification v.0.92. The model specifies the rules of the usage of servlets and JSP pages. According to this model JSP pages are responsible for presentation part, while servlets control data access operations and navigation in the application.

Essential components of Jakarta Struts are:

- servlet ActionServlet,
- class Action,
- class ActionForm and accompanying JavaBean subclasses,
- class ActionForward.

ActionServlet controls flows in the application. Receives application server requests (Tomcat, JBoss, etc.) and specifies, basing on a URI, which the Action class is supposed to deal with the request. The Action class verifies input data and may use the existing data sources.

In some cases, in order to serve a request, the information on passed values is required. Such information is passed by the ActionServlet to the appropriate JavaBean that is a subclass of the ActionForm class. The servlet, basing on a URI, makes a decision which ActionForm should be chosen.

HTTP requests are passed by the Action to the appropriate JSP pages. Paths to JSP pages are stored in the ActionForward class. While operations in a business layer are finished the Action class passes results to the servlet which finally generates a response (a web page) to the HTTP request. All the information required to serve a request, that is related to a specified URI, is taken from the ActionMapping class.

Information on Action, ActionForm, ActionForward and ActionMapping classes is stored in a file struts-config.xml which is read by the ActionServlet at the start. Basing on the information from the configuration file, the servlet builds an object database which is then called by all of the Struts components.

Jakarta Struts Framework is built upon the Model View Controller paradigm. Delivering the controller layer it fills the gap existing in the domain of web applications based on Java technology. The main advantages of Jakarta Struts are:

- delivery of uniform design patterns,
- delivery of a source code and configuration files framework which significantly speeds up projects development, especially in its initial stages,
- easiness to share work in teams while developing applications due to separation of a Java source code and a presentation part.

Detailed description of Jakarta Struts architecture may be found in [3].

3 Requirements Specification for Analysis and Design

The authors, experienced in the Struts-based web application development, distinguished five models that present simplified views of a system:
3.1 Web Application Model

This is the most general model. It presents the fundamental division of the system into smaller subsystems. Every subsystem corresponds to a set of use cases.

3.2 Use Case Model

The Use Case Model is useful while functional requirements of the system are formulated. Use Case packages are derived from the application subsystems. Every use case identifies one functional requirement. It is defined by an appropriate flow diagram.

3.3 Flow Model

This model demonstrates control flows between web application components. It consists of actions and JSP pages. Static aspect of each action is presented by a class diagram. The construction of every JSP page is described in the presentation model.

3.4 Structure Model

The Structure Model presents the structure of the actions that can be realized in an application. It uses class diagrams. Among the classes, typically it is possible to distinguish business logic classes, data wrappers, classes that control forms and classes that realize control flows.

3.5 Presentation Model

The Presentation Model shows the construction of every JSP page. The page is presented as a template with specified template items that fill the template.

4 UML Profile for the Development of Jakarta Struts Framework Based Web Applications

4.1 UML Extension Mechanisms

There are standardized mechanisms of UML allowing its extensibility. Due to them it is possible to adjust the set of UML modeling elements to the specific requirements of any modeled domain [2], [4].

The fundamental UML extensibility mechanism is a stereotype. Its role is to define new modeling elements as subclasses of existing UML metaclasses, together with their metaattributes and new semantics. The elements newly introduced have to be consistent with the standard UML semantics. Their role is to precise UML semantics to the specific requirements. The stereotype is defined by its name, base class (UML metaclass), graphic icon (optional) and semantics.
In order to specify properties of modeled elements, tagged values are used. Tag definitions introduce new types for tagged values and are defined in the form of embraced in parentheses, comma-separated strings of name:type pairs. The concrete values of properties are written in the form of embraced in parentheses, comma-separated strings of name=value pairs, where a name is derived from the proper definition of the tagged value. Another UML extension means are constraints, used to precise the semantics of modeled elements. They are mainly expressed in the specialized language, Object Constraint Language (OCL), but also using any mathematical notation, natural or programming languages. The constraints have the form of logical expressions, which are true only when the model is well-formed.

The modeling elements, defined for specific purposes, are usually grouped into profiles, i.e. stereotyped packages extending the UML metamodel. In the article the authors present the UML profile for the domain of analysis and design of Jakarta Struts Framework based web applications.

4.2 Summary of UML Profile Elements

The set of UML modeling elements, being used in the five models presented in the chapter 3, is summarized in a table 1. In the table, except for the name of an element, its base class and textual and (optionally) graphic notation are specified.

4.3 Modeling Elements Semantics

Web Application Model elements

WebApplicationModel. The characteristic of common web applications is that they comprise the set of typical functional modules, like information module (public), intranet module (private), Content Management System (CMS), internet shop module, etc. The web application model demonstrates the web application as the set of such functionally independent subsystems. It is the most general view of the application. The model comprises WebApplication package, which in turn comprises WebModule packages corresponding to functional subsystems of the web application. The dependencies between WebModules and packages of corresponding use cases are indicated as relationships of refinement.

WebApplication. The highest level package in the web application model. It groups functionally independent modules of the web application.

WebModule. The package that represents the functionally complete module of the web application. The set of functions it realizes is gathered in the corresponding use cases package. The dependency between WebModule and the corresponding use case package is indicated as relationship of refinement.

Use Case Model elements

UseCaseModel. The use case model presents the functions of a web application available for its users. The elements of this model, their notation and semantics are invariable in relation to [2]. Additionally, in the model the relationships between use cases and packages from the flow model corresponding to them are indicated. It is suggested to make use of the rules of specifying use cases described thoroughly in [5].

1 The graphic notation of elements is derived from [1]
## Table 1. The set of modeling elements, their base classes and notation

<table>
<thead>
<tr>
<th>Element name</th>
<th>Base Class</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebApplicationModel</td>
<td>Model</td>
<td>package stereotyped &lt;&lt;webApplicationModel&gt;&gt;</td>
</tr>
<tr>
<td>WebApplication</td>
<td>System</td>
<td>package stereotyped &lt;&lt;webApplicationModel&gt;&gt;</td>
</tr>
<tr>
<td>WebModule</td>
<td>Subsystem</td>
<td>package stereotyped &lt;&lt;webModule&gt;&gt;</td>
</tr>
<tr>
<td>UseCaseModel</td>
<td>Model</td>
<td>package stereotyped &lt;&lt;useCaseModel&gt;&gt;</td>
</tr>
<tr>
<td>UseCasePackage</td>
<td>Package</td>
<td>package stereotyped &lt;&lt;useCasePackage&gt;&gt;</td>
</tr>
<tr>
<td>FlowModel</td>
<td>Model</td>
<td>package stereotyped &lt;&lt;flowModel&gt;&gt;</td>
</tr>
<tr>
<td>FlowPackage</td>
<td>Package</td>
<td>package stereotyped &lt;&lt;flowPackage&gt;&gt;</td>
</tr>
<tr>
<td>JSP</td>
<td>Class</td>
<td>class stereotyped &lt;&lt;jsp&gt;&gt; or an icon:</td>
</tr>
<tr>
<td>Form</td>
<td>Class</td>
<td>class stereotyped &lt;&lt;form&gt;&gt; or an icon:</td>
</tr>
<tr>
<td>Action</td>
<td>Class</td>
<td>class stereotyped &lt;&lt;action&gt;&gt; or an icon:</td>
</tr>
<tr>
<td>StructureModel</td>
<td>Model</td>
<td>package stereotyped &lt;&lt;structureModel&gt;&gt;</td>
</tr>
<tr>
<td>StructurePackage</td>
<td>Package</td>
<td>package stereotyped &lt;&lt;structurePackage&gt;&gt;</td>
</tr>
<tr>
<td>PresentationModel</td>
<td>Model</td>
<td>package stereotyped &lt;&lt;presentationModel&gt;&gt;</td>
</tr>
<tr>
<td>PresentationPackage</td>
<td>Package</td>
<td>package stereotyped &lt;&lt;presentationPackage&gt;&gt;</td>
</tr>
<tr>
<td>Template</td>
<td>Class</td>
<td>class stereotyped &lt;&lt;template&gt;&gt;</td>
</tr>
<tr>
<td>TemplateItem</td>
<td>Class</td>
<td>class stereotyped &lt;&lt;templateItem&gt;&gt;</td>
</tr>
</tbody>
</table>

## Table 2. Tagged values for WebApplication and WebModule packages

{ImplName : String} WebApplication and WebModule packages may have some implementation-specific names. Such names should be in the form of strings without any whitespaces or non-ASCII letters. If the name is not stated explicitly, it is assumed that it is the same as the name of the appropriate package.
UseCasePackage. Use case packages categorize use cases with respect to the web application modules.

| ImplName : String | UseCasePackage may have some implementation-specific name. Such name should be in the form of a string without any whitespaces or non-ASCII letters. If the name is not stated explicitly, it is assumed that it is the same as the name of the appropriate package. |

Flow Model elements

FlowModel. The flow model presents all possible navigation paths and, consequently, permissible flows of control between the sections of the web application. To achieve this, it makes use of the collaboration diagram.

For the sake of dynamic generation of part of WWW pages, it is required to present some specific activities, in Jakarta Struts Framework called actions, which support the generation process. Their main roles are to collect the necessary data from the model layer to be presented on the web page and to steer the flow of control. The actions are sets of Java classes, consistent with Jakarta Struts Framework. Their structure is presented in the structure model.

In Struts-based web applications users navigate through JSP web pages (jsp). Such pages are generated on the basis of templates, described in the presentation model. Among JSP pages the pages containing web forms (form) are distinguished due to their specific treatment in Jakarta Struts Framework.

FlowPackage. The flow package presents the navigation paths that exist between web pages of the particular use case.

JSP. The class that represents a JSP web page in the collaboration diagram. As Jakarta Struts Framework allows developers to build JSP pages based on generic templates (tiles library), hence the presentation model is introduced. Such model makes use of notions of a template and a template item to describe the structure and contents of JSP pages.

Form. The class that represents a JSP web page containing a web form in the collaboration diagram. Such a web page builds on template in the same way as other JSP pages. JSP pages containing web forms are distinguished among all JSP pages due to their input role for the web application (they are gathering data from the users); except for them, all other web pages has only output role (they are presenting data).

Action. The class that represents a fragment of web application business logic. It is responsible for the display of a proper JSP web page. Typically, actions gather data from the model layer and control flow of control accordingly to specific conditions (e.g. whether the user is authorized to access a particular web page or not). Their structure, in terms of Java classes, is presented in the structure model.
Table 4. Tagged values for the FlowPackage

{ImplName : String} FlowPackage as well as JSP, Form and Action classes may have some implementation-specific names. Such names should be in the form of a string without any whitespaces or non-ASCII letters. If the name is not stated explicitly, it is assumed that it is the same as the name of the appropriate package or class.

Structure Model elements

StructureModel. The structure model presents the static aspect of the web application business logic, which is responsible for the display of a proper JSP web page. It demonstrates the relationships between Java classes constituting the action. At present there are no stereotypes introduced among the set of Java classes, even if they might be categorized as navigation controllers, data wrappers, data collectors, etc. However, their roles are described with specific prefixes added to their names.

StructurePackage. The structure package groups fragments of web application business logic, which are responsible for the display of proper JSP web pages. In other words, it comprises Java classes realizing the particular action.

Table 5. Tagged values for the StructurePackage

{ImplName : String} StructurePackage may have some implementation-specific name. Such name should be in the form of a string without any whitespaces or non-ASCII letters. If the name is not stated explicitly, it is assumed that it is the same as the name of the appropriate package or class.

Presentation Model elements

PresentationModel. The presentation model presents the inner structure of JSP pages using templates. This approach is supported by the tiles library, part of Jakarta Struts Framework. JSP pages usually comprises few conceptual blocks (template items), like header, footer, body, menu, etc. Some of them are invariable for all web pages of the web application (eg. a footer). The templates of web pages define where and which of template items should be placed in order to compile a complete web page. It allows to avoid unnecessary redundancy of code and makes the JSP code more manageable.

PresentationPackage. The presentation package groups templates and template items, which together compile a complete web pages.

Template. The class that represents the template of a JSP page in the presentation diagram. In Jakarta Struts Framework, the template is a part of configuration file. It states precisely where and which of template items should be placed for a particular web page. Additionally, there is always a generic JSP file, usually the same for all templates of web pages from the same web module, defining the structure of such web pages.
TemplateItem. The class that represents a fragment of a JSP web page. It is used to fulfill templates. In practice, a template item is a piece of JSP code written in JSP file.

Table 6. Tagged values for the PresentationPackage

<table>
<thead>
<tr>
<th>Tagged values for the PresentationPackage</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ImplName : String}</td>
</tr>
</tbody>
</table>

PresentationPackage as well as Template and TemplateItem classes may have some implementation-specific names. Such names should be in the form of a string without any whitespaces or non-ASCII letters. If the name is not stated explicitly, it is assumed that it is the same as the name of the appropriate package or class.

5 An Application Example

The elaborated extension will be presented on the basis of the documentation of the real web application. The application was realized by the authors by means of Jakarta Struts Framework, Java and JSP.

The examples demonstrating each of the introduced models are given below. The authors start from the analysis stage specifying functional modules and their functions. The last presented stage is the design stage where relation between actions and web pages are given To make the presentation more clear the diagrams were simplified.

Fig. 1. Functional modules (subsystems) of the web application

At the beginning of the analysis phase, functional modules of the web application were distinguished:
a public internet module – its content was available for all interested users,
a private intranet module – available only for authorized users,
a content management module.

Each of the distinguished modules is described by a corresponding use case diagram. A simplified diagram is shown in Figure 2: selected intranet functions (browsing meetings schedule and using bulletin board) available for authorized users are presented. Realization of the functions is described by corresponding flow diagrams. A selected use case realization is given by the collaboration diagram (Fig. 3). The diagram is highly simplified. It does not demonstrate, for example, commenting or browsing the contents of bulletin board messages.

The structure of each action that appear on the flow diagram is described by the class diagram. Example for an action bulletin board is given on Fig. 4. A typical realization of the action is done by four classes: a navigation control class, data management class, a wrapper class and a helper one for data base querying. Prefixes A, DM and D are used to emphasize a role of an appropriate class. The new stereotypes were not introduced in order to keep diagrams legible.

JSP pages are build upon templates basing on tiles library that is a part of Jakarta Struts Framework. Templates are in fact JSP files that include appropriate directives for JSP/HTML code enclosure. Templates may be filled gradually therefore the inheritance relation is presented on Fig. 5.

6 Summary

In the paper the UML profile supporting development process of Jakarta Struts Framework based web applications has been presented. Together with the profile, five models
The structure of the forum action is described in the structure package "Forum".

The structure of the JSP web page is described in the presentation package "Forum".

**Fig. 3.** A part of the structure diagram – an action that prepares JSP page of the bulletin board

**Fig. 4.** A part of the flow diagram (bulletin board function)
Fig. 5. A part of the presentation diagram (a page with the bulletin board messages)
describing different stages and viewpoints of analysis and design of Struts-based applications have been introduced. In order to prove the usefulness of abovementioned approach, selected fragments of the real world web application have been included. The future improvements to the profile and models are envisaged. The introduction of new models, presenting data-related aspects of a web application, like data flow model or data structure model, is being considered.

References

6. Working Material from the School on Natural Computation, SNAC
8. Reima Suomi, Paul Jackson, Laura Hollmén and Mats Aspnäs (Eds.), Teleworking Environments, Proceedings of the Third International Workshop on Telework
9. Robert Fullér, Fuzzy Reasoning and Fuzzy Optimization
10. Wolfgang Weck, Jan Bosch, Clemens Szyperski (Eds.), Proceedings of the Third International Workshop on Component–Oriented Programming (WCOP '98)
11. Abstracts from the 10th Nordic Workshop on Programming Theory (NWPT'98)
13. Christer Carlsson and Franck Tétard (Eds.), Intelligent Systems and Active DSS, Abstracts of the IFORS SPC-9 Conference
15. Tero Harju and Iiro Honkala (Eds.), Proceedings of the Seventh Nordic Combinatorial Conference
16. Christer Carlsson (Editor), The State of the Art of Information System Applications in 2007
17. Christer Carlsson (Editor), Information Systems Day
19. Reima Suomi, Jarmo Tähkäpää (Eds.), Health and Wealth trough Knowledge
20. Johan Lilius, Seppo Virtanen (Eds.), TTA Workshop Notes 2002
21. Mikael Collan, Investment Planning – An Introduction
24. Pirkko Walden, Stina Störling-Sarkkila, Hannu Salmela and Eija H. Karsten (Eds.), ICT and Services: Combining Views from IS and Service Research
25. Timo Järvi and Pekka Reijonen (Eds.), People and Computers: Twenty-one Ways of Looking at Information Systems
26. Tero Harju and Juhani Karhumäki (Eds.), Proceedings of WORDS'03
27. Mats Aspnäs, Christel Donner, Monika Eklund, Pia Le Grand, Ulrika Gustafsson, Timo Järvi and Nina Kivinen (Eds.), Turku Centre for Computer Science, Annual Report 2002
29. Mats Aspnäs, Christel Donner, Monika Eklund, Ulrika Gustafsson, Timo Järvi and Nina Kivinen (Eds.), Turku Centre for Computer Science, Annual Report 2003
30. Andrei Sabelfeld (Editor), Foundations of Computer Security
31. Eugen Czeizler and Jarkko Kari (Eds.), Proceedings of the Workshop on Discrete Models for Complex Systems
32. Peter Selinger (Editor), Proceedings of the 2nd International Workshop on Quantum Programming Languages
34. Kai Koskimies, Ludwik Kuzniarz, Johan Lilius and Ivan Porres (Eds.), Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language, NWUML’2004