Towards Systematic Model Assessment

Ruth Breu and Joanna Chimiak-Opoka

University of Innsbruck, Institute of Computer Science,
Techniker Str. 21a, Innsbruck, Austria
{Ruth.Breu, Joanna.Opoka}@uibk.ac.at
http://qe-informatik.uibk.ac.at/

Abstract. In this paper a novel approach for the tool–based quality assurance of models is presented. The approach provides a meta model framework for domain specific and tool–independent quality assessment in heterogeneous model landscapes. In our framework we provide the concepts of queries, checks and views defined on meta model level and interpreted over the whole model landscape. Queries, checks and views are described in a predicative language based on the structures of the meta model.

1 Introduction

After years of intensive standardisation activities in the context of UML (Unified Modelling Language) and the development of methods and tools, models have found their way to real software development. More and more companies and organisations use models to fill the gap between informal textual descriptions of requirements and the realizing code. Usage scenarios of models range from the analysis of business processes and system requirements to the documentation of software architectures and model driven software development.

Not surprisingly the use of models in real applications reveals new requirements and challenges. One of these challenges is the **quality assurance** of the models developed. Complex model landscapes (sets of related models) as e.g. developed within software architecture documentation in general contain inconsistencies and gaps. Quality assurance of these model landscapes cannot be done by pure manual inspection or review but requires tool assistance.

Drawing an analogy to quality assurance of code one can identify at least two important sub–disciplines of model quality assurance: model testing and static analysis of models.

Model testing can be applied in cases where the models are attached with a kind of executability—like in model driven software development or model simulation. We will not deal with this aspect in this paper and refer to [1] for a testing approach in the context of workflow models.

In this paper we address the **static analysis aspect of models**. Modern static code analysis deals with the quantitative analysis of dependencies within the code and is intimately connected with the notion of code metrics [2]. Analogous approaches to static model analysis can be found in the context of UML

J. Akoka et al. (Eds.): ER Workshops 2005, LNCS 3770, pp. 398–409, 2005.

diagrams [3]. However, the proposed indicators of high quality models, such as for example general diagramming metrics (e.g. number of elements of diagrams, number of stereotypes of diagrams) or diagram–specific metrics (e.g. for class/package diagrams: depth of inheritance hierarchy, number of child classes, number of parameters), are far less accepted than indicators of high quality code.

In this paper we present a novel approach for model assessment which goes into a different direction. Our first observation is that the quality of a model landscape is mainly influenced by the **interplay of model elements and diagram types**. For example, consider a requirements specification consisting of the following set of models: use case model, scenarios referring to the use cases, class model, state diagrams and schematic use case descriptions. All these models are based on a set of *model elements* like use case, actor and business class. The use of these model elements in different models creates **complex interrelationships** and multiple sources of inconsistency. In the example, the initiating actor in the use case description has to be an actor related with the use case in the use case model. The model elements and their interrelationships cannot be defined in a general way for all kinds of UML diagram types but depend on the underlying method and application context.

A second observation is that in many cases the quality checks of a model landscape cannot be done in an automatic way but the quality manager can be supported by *views* that provide **aggregated information** about the model landscape. For instance, a view on an enterprise model may list all information objects and applications together with the information which information object is related with which application in the model. This information can be generated from the business process model and enables the quality manager to perform cross–checks on the model landscape. In general we distinguish *queries*, *checks* and *views*. Queries are functions on the current status of the model landscape returning some value or model element, checks are queries with Boolean result. Queries may be embedded in views that represent queries for multiple input.

The applications of our approach are many–fold contexts in which complex model landscapes are developed. This ranges from requirements specification to the documentation of software architectures and enterprise models. In particular, the concepts presented in the sequel are aligned with the requirements of three of our cooperation projects with industrial partners (MedFlow—Quality Assessment of Business Processes in Health Care, ProSecO—IT Security Assessment in Enterprises, Pro2SA—Model–Based Strategic Alignment).

An important further aspect coming out of these applications is the **heterogeneity of the model landscape**. We want to reason about model elements which are defined in (UML–)tools, semi–formal or formal text documents or even code. As a common syntactic framework we make therefore use of XML (Extensible Markup Language).

The sequel of the paper is structured as follows. In section 2 we present related work and give more information about the application context in our projects. Section 3 introduces the basic framework and concepts and in section 4 we introduce the concepts of queries, checks and views. Section 5 draws a conclusion.

2 Related Work

There are several approaches, mostly in industrial contexts that deal with quality checks of model landscapes. In the ARCUS project [4] in cooperation of the Bayerische Landesbank and sd&m a meta model for describing IT landscapes has been defined. An important element of this meta model are relationships between the model elements that define a notion of traceability. A plug–in for an UML tool supports a fixed set of queries over the model landscape. In a study at BMW [5] a set of quality checks for model landscapes that are input to an in–house MDA (Model Driven Architecture) tool has been developed. As further example Siemens Princeton has developed a tool for checking requirements specifications [6]. For the implementation these approaches use the scripting facilities or programming interfaces of UML tools or graphic programs such as Rational Rose, Together, Visio or Adonis [7, 8]. These three examples are indicators of the practical relevance of the topic. Our approach goes one level beyond in the respect that it provides a generic platform for describing checks and views over a given set of interrelated model elements.

Concerning the syntactic and semantic framework we use parts of the OCL (Object Constraint Language, v. 2.0) for describing navigations over model elements. Existing methods and tools are rather dedicated to homogeneous environments. For example the Executable UML [9] can be used to execute and test models defined in UML. This is enabled by a formal action semantics for models (in an action language) and specification of constraints tags (in OCL). And with the OCLE tool [10] it is possible to check UML models against well–formedness rules, methodological, profile or target implementation language rules expressed in OCL. It is also possible to obtain metric information about UML models. This tool also uses XML, but only as a data tier, while we use XML also for modelling purposes. Our method provides mechanisms to carry out checks for domain specific models defined in heterogeneous environment.

In the subsequent chapters we present the basic concepts of QUARC (QUAlity Requirement Checks). Currently we work on a tool–based realisation which will be presented in accompanying papers. QUARC is aligned with the requirements of three of our cooperation projects: MedFlow, ProSecO, and Pro2SA. In context of our projects the focus of QUARC is the automatic check of model consistency and the generation of views for supporting manual checks, e.g. concerning media or applications ruptures and appropriate tool support of the actors (MedFlow); the generation of aggregated views in a highly linked heterogeneous model landscape (ProSecO); information aggregation in model landscapes and evaluation of quantitative data associated with model elements (Pro2SA).

Although the case study presented in the subsequent sections has been taken out of the MedFlow project, the method is not specific for clinical process and it can be also applied to models from other domains (e.g. industrial, commercial, enterprise). In MedFlow we develop an approach for the systematic quality check of models describing business processes in health care. The background of this project is the task of targeting standard hospital information systems towards the needs of complex organisational processes in hospitals. More about this project can be also found in [11].

3 Basic Concepts

In this section we introduce the basic concepts and model packages our approach is based on. Because the concepts are tightly correlated, we used arrow symbols for cross–definitions (→ *definition*).

Model. A model (at instance level) is a structured document that is subject of the quality assessment. We consider any type of UML models like class diagrams and sequence diagrams, but also text documents or code. Models at instance level are based on → *meta model element*s to describe properties of systems. Each model at instance level has an associated → *model type*.

Meta model. The meta model defines the universe of discourse for the quality assessment. The meta model is a class diagram modelling the → *model element*s and their relationships. This class diagram is contained in the → *meta model package*. A meta model may be associated with a specific method (e.g. use–case based requirements specification), a particular application domain (enterprise models, embedded systems) or with a particular development environment (e.g. a meta model associated with an MDA–tool).

Meta model element. A meta model element is a class in the → *meta model* describing a basic concept used in the → *model*s at instance level. Examples for meta model elements are *actor*, *information resource*, *business process*, *action* and *logical tool*. Meta model elements are the basic units over which queries and views can be formulated. Meta model elements may have attributes (e.g. the medium of an information resource) and may be linked with other meta model elements. These links have to be directed (in either or both directions) indicating where the → *model type package* link is maintained.

Meta model package. The meta model package contains the → *meta model*, the → *EM–mapping*, the → *model type package*, and the queries, checks and views.

Model Type. A model type groups models in a category. A model type (at instance level) is subject of the quality assessment. The model type characterises the role of the model within the underlying method. For instance, we may have model types *business class diagram* and *technical class diagram* in a context where we assess the documentation of software architectures. The interdependencies of the model types are described in the → *model type package*.

Model type package. The model type package contains a class diagram describing the interrelationships between the → *model type*s. We call an instance of this class diagram a **model landscape**.

EM–mapping (Meta Model Element–Model Type Mapping). The EM–mapping maps the → *meta model element*s to the → *model type*s defining in which model type the meta model elements are defined and used. As an example, the model element *information* is defined in the *Information Model* and used in the *Business Process Model*.

In the sequel we provide an example together with more detailed information about the basic concepts. The description of queries, checks and views is introduced in section 4.

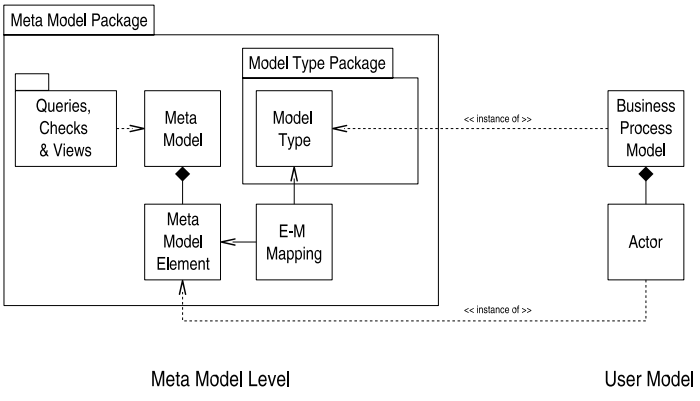

Fig. 1. The notions on meta model level and user model

Figure 1 summarises the notions at meta model level and user model level together with their interrelationships. The two levels correspond to the M2 layer models (Meta Model Level) and M1 layer models (User Models) defined in the MOF (Meta–Object Facility) Metadata Architecture.

3.1 Meta Model

Example 1. Figure 2 depicts (a portion of) the meta model for the quality assessment of clinical process descriptions. Processes are defined in a hierarchical way based on the notion of sub–processes and actions. Each action is associated with the executing actor, input and output information and logical tools supporting the executing actor.

3.2 Model Type Package

The model type package describes the model landscape that is subject of the quality assessment. The model types are related with «uses» relationships which means that model elements of one model type are used in the other model type. Aggregation is used for a hierarchical structuring of model types.

Example 2. Figure 2 depicts the set of model types used in our case study.

Each model type is associated with a type which is either an UML diagram type or XML, i.e. we assume non–UML models to be interconnected via XML structures.

Example 3. The following models, from Example 2, are defined in UML: *Business Process Model* as activity diagrams, *Organisation Model* and *Information Model* as class diagrams, *Logical Tool Model* as component diagrams and finally *Physical Tool Model* as deployment diagrams. The other models are defined in XML: *Description of Actions* and *Permission Model*.

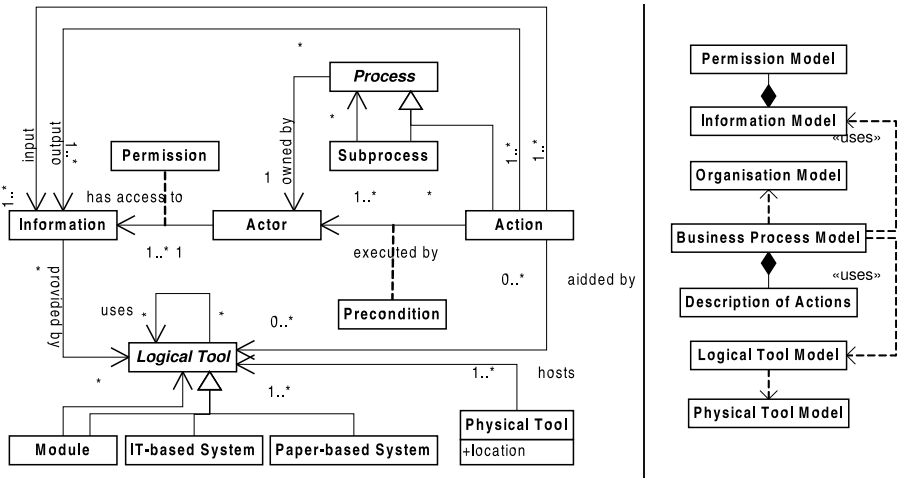

Fig. 2. A meta model (on the left) and model types (on the right) for clinical process

For each model type we assume an XML–representation. We have chosen XML because it is a standard notation and already supported in the UML–context by XMI (XML Metadata Interchange) as a standard format for model interchange. The most important features of XMI are its built–in nesting mechanism and the possibility of transformation from OCL–navigation–expressions to XPath–expressions.

In our case study the information model is an UML class diagram interconnected via its XMI representation, whereas the description of actions is a text document with an XML interface. The action description contains an informal description and information about the executing actor, input and output information and the logical and physical tool used in this action.

Example 4. Figure 3 depicts an action description at instance level. The XML description is based on our XML Schema for action description (due to limited space we do not present its full syntax here), in which the following attributes of an action are defined: `name` of the action, `role` involved in execution of it, `input` and `output` information needed or produced by it, `tool`, both logical and physical,

```
<action name="check␣of␣patient's␣data">
  <role name="control␣station"/>
  <input>
    <information name="referral">
  </input>
  <tool logical="MEDAS␣(KIS)" physical="PC-LST-1"/>
  <tool logical="Power␣Chart␣(KIS)" physical="PC-LST-1"/>
  <tool logical="RIS"  physical="PC-LST-2"/>
</action>
```

Fig. 3. Sample Ation description

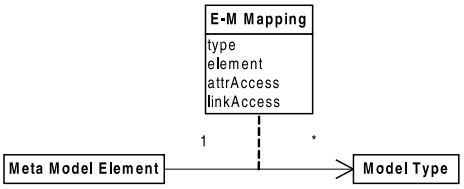

Fig. 4. Mapping between abstract and concrete level

used for execution of it and some other properties. In this example we deal with `check of patient's data` action executed by `control station` role, which needs `referral` as an input information and three tools to be completed.

3.3 EM–Mapping

The EM–mapping provides the interconnection between meta model elements and the model types. More precisely we define for each meta model element,

- in which model type it is defined,
- in which model types it is used, and
- how attributes and outgoing associations can be retrieved.

Figure 4 illustrates the kind of information that is provided for each meta model element. The type is either *defined* or *used*, the element attribute maps the meta model element to some model element of the target model type. For UML diagram types this means that the meta model elements are mapped to elements of the UML meta model (in some cases the meta model may itself contain elements of the UML meta model). The attributes attrAccess and linkAccess define the access to the attributes and outgoing links of the meta model element at XML level, an aspect that is not treated in more detail in this paper.

Example 5. Table 1 depicts a part of the mapping for the meta model elements in our case study.

Table 1. Sample EM–Mapping (Schematic)

Meta Element	Type	Model Type	Element
Action	def	*Description of Actions*	XML::action.xsd::action
Action	use	*Business Process Model*	UML::Activity Diagram::Action
Information	def	*Information Model*	UML::Class Diagram::Class::Information
Information	use	*Description of Actions*	XML::action.xsd::action::input::information
Information	use	*Description of Actions*	XML::action.xsd::action::output::information
Logical Tool	def	*Logical Tool Model*	UML::Component Diagram::Component
Logical Tool	use	*Description of Actions*	XML::action.xsd::action::tool.logical
...	...	...	...

4 Queries, Checks and Views

Based on the structure of meta model elements in the next step a set of queries, checks and views can be defined. These are defined by method responsibles whose task is to assist developer teams in the systematic quality assessment of the models developed.

Query. A query is a function over meta model element instances returning a value (e.g. a Boolean value or an integer) or meta model element instance(s). The result of a query may depend on the input parameters and the network of currently existing instances of meta model elements in the model landscape. The goal of a query is to provide the modeller with information about the model landscape.

Example 6. Examples of queries in our case study may be the following: Amount of actors in the model landscape; The set of logical tools an actor is related with in the business process model (via the actor–action–logical_tool relationships).

Check. A check is a query with Boolean value as result. The goal of a check is to assess a model landscape based on a given constraint. Moreover, we associate each model landscape with a set of predefined (well–formedness) checks that are related with the model type package and the EM–mapping. The user models have to conform to the model structure described in the model type package. For instance, each model element that is *used* in the model landscape should also be $defined$ in some model (checking the consistency of *used* and $defined$ relations in the EM–mapping).

Example 7. An example of a check is the following: There exists at least one actor and one information class in the model landscape.
An example of a predefined check: Each action used in *Business Process Model* has to have a textual description in *Description of Actions*.

View. A view is a query whose result is represented for all (or a restricted set of) input elements and may be equipped with further information regarding the quality assessment of the result. The goal of a view is to present aggregated information over a model landscape and to support the modeller in model inspection. The benefit of the view is to support the modeller in a cross–check of the business process model.

Example 8. Table 2 depicts the example view `InformationInLogicalTool` listing all information types and logical tools defined in user models and indicating if the given information is related with the given logical tool. Here $Referral$, $Diagnostic\ Findings$ and $Image$ are classes in the *Information Model* (class diagram) and KIS, $PACS$, $PaterNoster$ are components in *Logical Tool Model* (component diagram). The result is defined by the OCL–like expression as given in Example 9 and is true for a given information and a given logical tool, if the information is saved in the tool.

Queries, checks and views in our approach are described by OCL based predicative language expressions that are constructed over the class diagram of the

Table 2. Sample View InformationInLogicalTool

Information	Logical Tool	Result
Referral	KIS	true
Referral	PACS	false
Referral	PaterNoster	true
Referral	...	...
Diagnostic Findings	KIS	true
Diagnostic Findings	PACS	false
Diagnostic Findings	PaterNoster	true
Diagnostic Findings	...	...
Image	KIS	false
Image	PACS	true
Image	PaterNoster	true
Image	...	...
...	...	...

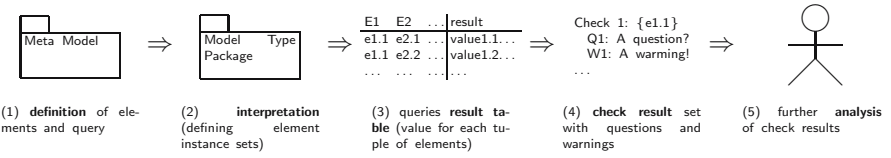

Fig. 5. The process of defining, interpreting and executing a view

meta model. These expressions are embedded into XML structures that provide the syntactical framework.

In Fig. 5 the process of defining, interpreting and executing a view is depicted. In the first step the view is defined over the meta model elements. Then in the second step the sets of instances of meta elements are collected from the corresponding user models (via the model database). The result values are calculated for each combination of elements of the given sets and in the third step the result table is presented. In the fourth step the result table is subject of further analysis and automatically executed checks. The result of the fourth step are sets of elements fulfilling the condition in the given check. For non–empty check sets questions and warnings may be shown to the user. The questions are used in checks, for which additional analysis of the result is needed. The warnings could be used in checks for well–formedness rules. In the fifth step the analysis of the questions and warnings is made by user.

In the sequel we will present in more detail the structure of queries, checks and views together with sample expressions.

4.1 Queries and Checks

Checks and queries are defined as functions over individual or aggregated elements. The formal definition of such a function is expressed as follows:

$$\mathcal{Q}(\mathbf{p}_1 : \mathcal{T}_1, \ldots, \mathbf{p}_n : \mathcal{T}_n) \quad \mathcal{T} : \mathcal{E} \tag{1}$$

where $\mathcal{Q}$ is a query or check name; $\mathbf{p}_i$ is an instance of meta model element $\mathcal{T}_i$; a result is an expression $\mathcal{E}$ of a given type $\mathcal{T}$. If type $\mathcal{T}$ is Boolean we call the function $\mathcal{Q}(\cdot)$ a **check**.

Example 9. A simple check could answer the question if a given information is saved in a given logical tool:

```
InformationInLogicalTool(i:Information, lt:LogicalTool) Boolean
      : i.logicaltool.select(name = lt.name).notEmpty().
```

4.2 Views

The views are built according to information from concrete models and formally we define them as follows:

$$\mathcal{V}(\mathcal{T}_1\ [\mathcal{F}_1], \ldots, \mathcal{T}_n\ [\mathcal{F}_n]) \quad \mathcal{T} \tag{2}$$

where $\mathcal{V}$ is a view name; $\mathcal{T}_i$ is one of n types (meta elements), which could be optionally filtered with a given filter $\mathcal{F}_i$.

Let $\mathcal{P}_i$ denote the set of instances of the given meta element $\mathcal{T}_i$ occurring in the user models. To complete the view we have to consider all tuples from the Cartesian product $\mathcal{P}_1 \times \cdots \times \mathcal{P}_n$. For each tuple $(\mathbf{p}_1 : \mathcal{T}_1, \ldots, \mathbf{p}_n : \mathcal{T}_n)$ we calculate the result $(\mathbf{r})$ using a query defined in the view $(\mathcal{Q}(\cdot) : \mathcal{T})$, thus $\mathbf{r} = \mathcal{Q}(\mathbf{p}_1, \ldots, \mathbf{p}_n)$. We extend the tuple adding the result and we therefore obtain extended tuples in form $(\mathbf{p}_1 : \mathcal{T}_1, \ldots, \mathbf{p}_n : \mathcal{T}_n, \mathbf{r} : \mathcal{T})$. The result of the view is a set of extended tuples.

Example 10. We define a view for informations and logical tools:

$$\mathcal{V}_{InformationInLogicalTool}(Information, LogicalToool)\ Boolean.$$

In this view we use the query `InformationInLogicalTool(·)`, as defined in Example 9. The result is a set of tuples:

```
Tuple(i : Information, lt : LogicalTool, r : Boolean),
      where r = InformationInLogicalTool(i,lt).
```

An example result of an evaluation of the view is shown in Table 2.

If we would like to consider only a subset of the input set of instances we have to apply a filter. The filter $\mathcal{F}_i$ defines a constraint for the set $\mathcal{P}_i$. A set with filter is defined as $\mathcal{P}_i = \{\mathbf{p}_i : \mathcal{F}_i(\mathbf{p}_i)\}$.

Example 11. If we would like to consider only physical tools (`pt`) located in the radiology ward then we use the following filter.

```
pt.location='Radiology'
```

Complementary Checks. Additionally we support the definition of complementary checks, questions and warnings within the view. Complementary checks are defined as queries over the set of tuples (result table). As a result of a complementary check we obtain the set of elements or tuples fulfilling the query.

Example 12. Let's say we would like to find unsaved information. We use the result of the view defined in Example 10 ($\mathcal{V}_{InformationInLogicalTool}$), and make a complementary check over this result. If we denote the result by `view:Set(Tuple)` then we can find all unsaved information using the following expression:

```
context view def:
collect (info : Information |
      self.select(i = info and r = 'true').size() = 0).
```

If we obtain a non–empty set as a result of complementary check, the warnings (see Example 13) or the questions (see Example 14) are shown. **Warnings** and **questions** are described in natural language and are not processed automatically. Warnings could be defined for checks over well–formedness rules.

Example 13. If the result of the complementary check defined in Example 12 is a non–empty set then the set will be listed and the warning *Each Information should have a medium!* will be shown.

Example 14. Let's say we would like to find redundant information, i.e. information saved in many logical tools. We use the result of the view defined in Example 10 ($\mathcal{V}_{InformationInLogicalTool}$), and make a complementary check over this result using the following expression:

```
context view def:
collect (info : Information |
      self.select(i = info and r = 'true').size() > 1).
```

If the result of the complementary check is a non–empty set then the set will be listed and the question *Is consistency of the redundant information warranted?* will be shown.

5 Conclusion

In the preceding sections we have presented a novel approach for the tool–based quality assurance of models. A main idea of this approach is to provide a meta model framework supporting application–specific quality assessment, tool–independent expression of quality assessment criteria and quality assessment in heterogeneous model landscapes both comprising (UML) models and textual models.

In our approach we provide the concepts of queries, checks and views. Queries are model retrievals, checks support automatic check of model constraints. Views support the modeller with aggregated information about the model landscape and may be associated with informal checks and heuristic quality indicators.

The approach presented is work in progress. Currently we both work on the final definition of an OCL based predicative language to describe queries, checks and views and on the software architecture of the related tool. Our work is driven by practical requirements of cooperation projects with industrial partners.

References

1. Breu, R., Breu, M., Hafner, M., Nowak, A.: Web service engineering—advancing a new software engineering discipline. In: Proc. of 5th International Conference on Web Engineering. (2005) (accepted).
2. Fenton, N.E., Pfleeger, S.L.: Software Metrics — A Rigorous and Practical Approach. Thomson, London (1997)
3. Gronback, R.: Model validation: Applying audits and metrics to uml models. In: Proc. of Borland Conference. (2004) (available on http://bdn.borland.com/borcon2004/).
4. Heberling, M., Maier, C., Tensi, T.: Visual Modeling and Managing the Software Architecture Landscape in a Large Enterprise by an Extension of the UML. In: Second Workshop on Domain-Specific Visual Languages. An OOPSLA Workshop, Seattle, WA (2002)
5. Jug, F.: Methods and techniques for quality assurance in software development process in bmw group (in german). Master's thesis, Technical University of Munich, Dep. of Computer Science (2004)
6. Berenbach, B.: Evaluating the quality of a uml business model. In: Proc. of 11 IEEE International Requirements Engineering Conference, Monterey, CA, USA (2003)
7. Junginger, S., Kuehn, H., Strobl, R., Karagiannis, D.: The next generation business process management toolkit ADONIS (in German). In: Wirtschaftsinformatik. Volume 42. University of Trier (2000) 392–401
8. BOC: Adonis. http://www.boc-eu.com/advisor/adonis.html (2000) access 2005-04-24.
9. Mellor, S.J., Balcer, M.J.: Executable UML. A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
10. LCI team: Object constraint language environment (2005) Computer Science Research Laboratory, "BABES-BOLYAI" University, Romania.
11. Saboor, S., Ammenwerth, E., Wurz, M., Chimiak-Opoka, J.: Medflow—improving modelling and assessment of clinical processes. In: Proc. of 19th Medical Informatics Europe, MIE 2005. (2005) (accepted for oral presentation).