An Active Approach to Model Management for Evolving Information Systems

Henrik Gustavsson[1], Brian Lings[2], and Bjorn Lundell[1]

[1] University of Skovde, Department of Computer Science
P.O. Box 408, SE-541 28 Skovde, Sweden
{Henrik.Gustavsson, Bjorn.Lundell}@ida.his.se
http://www.his.se/ida/

[2] University of Exeter, School of Engineering and Computer Science
Prince of Wales Road, Exeter EX44PT, UK
B.J.Lings@dcs.exeter.ac.uk

Abstract. It is desirable to be able to interchange design information between CASE tools. Such interchange facilitates cooperative development, helps in avoiding legacy problems when adopting new tools, and permits the use of different tools for different life-cycle activities. Exchanging model transformation information is particularly demanding in the context of cooperative maintenance of evolving systems. In this paper we suggest an approach using active transformation rules. We show how transformation rules can be expressed using a modest extension of the Object Constraint Language of the UML standard, and actively interpreted. The approach allows existing UML-based tools or repository systems to be readily extended to actively manage models in evolving information systems.

1 Introduction

In CASE tools in general, and in repository systems in particular, it is desirable to be able to interchange information between different tools in a toolset [4], [9]. Such interchange can facilitate cooperative development, through the exchange of design documents. It can also help in preserving design information when a new tool is adopted. Under some circumstances it may also permit the use of different tools for different life-cycle activities.

Many CASE tools today use mappings or design transformations to automate different tasks within an IS lifecycle design process. Few tools, however, support user definable design transformations, and those that do use proprietary languages for their definition [13]. Interchanging a set of models for the purpose of cooperative development can therefore be problematic. In particular, if the interchange is between tools that do not support the same set of design transformations, inconsistencies will be introduced. We argue, therefore, that design transformations must themselves be interchanged, and in such a way as to facilitate their use in an importing tool.

What is required is an architecture which will allow efficient execution of design transformations in a tool, and the export of both transformations themselves and details of their usage in a design, for example in transforming a conceptual model into a logical one. The set of interrelated design models could then be successfully manipulated using an importing tool, without it previously having been configured to handle the transformations involved. Such extensibility in the IS modelling area is the goal of the research reported here.

1.1 Transformations

In the area of meta modelling and repository systems, adoption of the UML [26], MOF [27] and XMI [25] standards has made it possible to interchange information and to guarantee extensibility in meta modelling systems [9]. A number of implementations of repository systems that currently support UML also support XMI interchange of data, including the France Telecom model repository tool [1] and the SPOOL design repository [19]. These projects support extensible, standards based meta models. In addition to supporting an extensible meta model, the Microsoft Repository [3] also supports the expression of transformations. It uses a model called the OTM (open transformation model) to define and store transformations. The main purpose of this is to support transformations in data warehouses, so the model for transformations is affected by its roots within the data warehousing field. However, in most other cases, mappings and transformations (or *model management* features [2]) used in tools connected to repositories are neither interchangeable nor extensible.

The use of proprietary languages for transformations inhibits interchange between tools. The Microsoft Repository uses a proprietary combination of SQL and OLE software to execute transformations. The DB-MAIN project [13] mainly uses a procedural language with a constraint language to define pre and post conditions for transformations. A logic-based language has been proposed elsewhere [23].

Many different uses have been found for design transformations (see [12], [13], [15], [16], [23]). Most of these sources, however, do not discuss how the modelling decisions leading up to a design transformation are to be represented. Nor do they discuss how to represent the information required to perform a design transformation - not all relevant information is directly present in a model (for example, required parameters from the user). Other systems, such as the DAIDA project, use a knowledge-based approach to design transformations. This allows sophisticated deductions to be made from the knowledge base, for instance which transformations were used in mapping between models [17].

The DB-MAIN project, however explores the use of design histories as a means to represent design decisions [14]. The model before a transformation is not maintained in its entirety, but modelling decisions are indirectly captured through state transitions. Using this ‘history’, a model can be wound back to the time before a certain transformation was applied, so that the transformation can be reapplied with, for example, a different set of input parameters, resulting in a different destination model.

However, the use of design histories to achieve a higher degree of modelling transparency has a number of drawbacks. Firstly, a very sophisticated versioning system is required which supports multiple and branching histories. In addition, the system must be capable of inferring modelling decisions and properties from these histories. Secondly, it may in some cases be difficult to tell whether a certain change in a model is due to the application of a transformation or direct user intervention. This is because design histories serve a dual purpose, in that they support both historical information about previous versions of a model and an indirect representation of those modelling decisions. Knowledge based approaches, on the other hand, require sophisticated inference engines, and use proprietary languages and representation techniques which thus make them unsuitable for a scenario where interoperability between different tools is of prime importance.

In an earlier paper [10] we have suggested an approach by which design transformations can be freely interchanged between tools. Transformations are represented using a conservative extension of the OCL language. The approach is independent of proprietary languages and techniques. Further, since more and more tools are adding support for OCL [24], it will become increasingly straightforward to adopt this approach in existing tools. The approach in that paper, however, has a significant drawback: rule execution is independent of the event that triggers a transformation. This problem makes the approach less powerful and significantly less suitable for maintenance. In this paper we suggest how to remove this drawback, developing the OCL language to support reactive behaviour suitable for co-operative management of model evolution.

1.2 OCL as a Conceptual Language for Meta-modelling Constraints and Actions

Active databases have been proposed, over the years, for a wide variety of different tasks in many different application areas. In repository systems, active databases have been used to automate common tasks such as general model processing [18] and change management [8]. Such proposals, however, have relied in large part on platform dependent models or languages. This would impede successful application of the techniques to systems which do not share a common platform.

The Object Constraint Language [20] is part of the UML standard [26]. Amongst other things, it is used to introduce pre and post conditions to, and to place guards on, methods. The language is platform independent, declarative and efficient for querying and navigating object-oriented data. Even though the OCL language, as a conceptual language, is intended for object-oriented modelling, it can be used with other forms of modelling. It can, for instance, be used to specify constraints on SQL databases [7].

This versatility makes OCL a good language for platform independent specification of conditions in repository systems and meta models.

An extension to the OCL language to support actions [21] has been put forward, but the proposal stopped short of suggesting that such actions be executed using an OCL interpreter. However, to achieve a high degree of modelling transparency [5], the simultaneous update of interrelated parts of dependent models is needed. The most common way to achieve this is by transforming a model into one which reflects the required changes – that is, active behaviour.

1.3 A Novel Approach to Modelling Transparency

This project takes a novel approach to increasing modelling transparency in that transformation patterns representing modelling decisions made by a user are represented explicitly as part of the modelling information stored in a repository. The repository thus directly represents transformations, the parameters needed to perform each selected instance of a transformation, and the results of such transformations (in the form of updated models). In order to completely support the desired increase in modelling transparency, the objects that result from a transformation are also connected to the source objects using ordinary associations. This allows a connection to be navigated, for instance to allow a tool user to find the set of relational tables that result from the transformation of an entity type.

In a previous paper we have shown that transformation rules can be expressed using a conservative extension of OCL. In this paper we show how a tool can be made to react to state changes in its meta modelling repository through the addition of events to transformation rules. A further modest extension to the OCL language is proposed, to support context variables to receive parameters from event occurrences. We have tested the ideas through the implementation of an active repository system with an event detector and rule manager suitable for model management in a UML environment. The proof of principle system used to test the examples used in the paper is available on request (henke@ida.his.se).

2 Overview of Approach

Although the approach outlined is designed to be generally applicable for multi-model management, our chosen application context is CASE data interchange for cooperative design. We believe it is beneficial in such contexts to support the active interchange of design transformations.

In the general approach, each design transformation is represented by a set of rules which, given specific model and parameter information, can be used to bring about that transformation. This offers better support for the incremental update of models typical of cooperative design. The OCL language has been chosen to represent design transformations[1]. However, OCL traditionally supports neither updates nor active behaviour. Other authors have suggested extensions to OCL for introducing active behaviour [21]. In our work, we extend the OCL language and its interpreter to allow the expression of active behaviour with update. In general, the fundamental issues to be addressed in moving to active behaviour are [28]: event specification and detection, access to context information by rules, and access to state transition information.

The proposed extension uses an ECA (Event Condition Action) format for rules to provide support for active design transformations. The use of ECA rules gives a number of benefits over a condition action based approach [6]. Firstly, events and conditions play different roles in the system, allowing the repository to react directly to state changes in the context in which they occur. Making the event explicit thus allows finer grained control over when execution occurs. This increases flexibility in execution semantics. This latter is important, since this project strives to be as platform independent as possible so that the ideas in the approach can be adapted to fit existing tool environments. There is also a performance benefit in that by using an event to trigger a condition check, fewer conditions have to be evaluated for the same database state.

2.1 Meta-model Extensions for Active Behaviour

There are different ways in which reactive behaviour can be supported, each placing different requirements on the rule scheduler and event detection mechanism. One very important consideration for this project is the ease with which the techniques can be incorporated into existing CASE technology. The suggested methodologies should thus be simple enough to be easily implemented in existing tools or tool infrastructures.

It has only been found necessary to include primitive database events in the rule system; no transformations so far studied have required the introduction of temporal or composite events. Hence, the rule system proposed only recognises events that occur when objects are inserted into the model, when objects are updated in the model, and when objects are deleted from the model. However, it has been found useful to distinguish a separate set of event types for the modification of collections, i.e. the creation, deletion and update of associations between objects in models.

In order to support the specification of active OCL rules, it has been necessary to extend the OCL rule language beyond that suggested in our previous work [10] by the addition of a list of events that can trigger a transformation rule:

```
Contextclass: <Context class specification>
Event: <event specification>
Condition: <condition specification>
```

[1] An explanation of the rationale behind the choice of this language can be found in [8].

```
Declaration: <declaration specification>
Action: <action specification>
```

In order to be able to represent transformations directly within the models, some kind of meta modelling support is necessary. In order to achieve a high degree of interoperability this model has been kept as simple and as generic as possible, in contrast with, for example, the Microsoft Repository approach [3] that models transformations with proprietary and domain specific structures. A number of superclasses have been proposed previously [10] for this purpose in a non-active environment; these can be inherited by the other meta model classes. The only extension required is the addition of an event property to the rule class.

2.2 Event Types and Context Variables

The behaviour of an active set of models may be heavily dependent on the cascading of rules (if multiple levels of models are used). It has been found useful to differentiate between primary events and secondary events. Primary events occur as a direct result of user interaction, for example check-in of a model or direct modification of a modelling object. Secondary events occur when the model contents are modified by a transformation rule. By differentiating between the six primary event types and the six secondary event types, it is possible to perform different actions depending on whether an update came from inside or outside the repository. This gives increased control over rule cascades since a rule, through its triggering event, has knowledge of whether it is executing as the consequence of a cascaded event or as a direct result of an update by a tool user. For example, it allows increased control over cycles in rule cascades.

In active databases, context variables (event parameters [6]) contain information about the state of the data before and after updates, so that rule behaviour can be expressed in terms of state change. In order to support this type of behaviour for the active set of models it is necessary to extend OCL to allow the use of context variables in transformation rules. In earlier work [10], we introduced updates to the OCL language. In particular, the aliasing mechanism present in the OCL standard definition was used to allow new objects to be referenced, and thus be modified in different ways. In order to limit the scope of the necessary changes to an OCL interpreter, it would be beneficial to further use the aliasing mechanism in supplying context variables.

In the prototype implementation, user updates to the repository are taken to be atomic actions. Transformation rules are therefore executed only after the metadata updates corresponding to these atomic actions. We have introduced two new global aliases for use as context variables. They work in the same way as the “SELF” alias, which identifies the context object. The “OLD” alias identifies the object state before the triggering update, and the “NEW” alias the object state after the triggering update. These context variable aliases have slightly different meanings depending on the type of event triggering the rule using the aliases (see table 1).

Table 1. Event types and context variable descriptions

Event	Alias	Description
Insert	NEW	Provides access to the values of the newly inserted object. Since the object is new, there is no “OLD” variable.
Delete	OLD	Provides access to the object as it was before deletion.
Update	OLD NEW	Provide pointers to the information before and (respectively) after the update.
Collection Insert	NEW	Provides access to the object which has been added to the collection.
Collection Delete	OLD	Provides access to the object which is going to be removed from the collection.
Collection Update	OLD NEW	Only occurs for 0..1 or 1..1 cardinalities. In this case, the “OLD” alias provides access to the object in the collection before the update, and the “NEW” alias provides access to the object in the collection after the update.

Since the OCL language displays collections and properties in the same way, but since the actual handling of associations and properties must account for conceptual differences, this separation simplifies the management of these two types of event.

A specific update collection event type is also introduced for collections that contain at most one object. When this is changed so that it identifies some other object, an update collection event is fired, indicating an update of an existing collection. The main difference between collection events and property events is that in property events, the object in the new and old variables is of the context class type. In contrast, for collection events, the new and old variables refer to the object that was added/removed from a collection, which may not be of the same type as the context class.

3 Example of Approach

To allow easy comparison with existing approaches, an example rule set has been created which demonstrates some important types of transformation handled in the related work cited earlier (section 1.1). The set of example transformation rules and the example meta model (see figure 1) are not intended to be complete, but are designed to show the characteristics of an active approach. We have successfully captured the design transformations used in the real-world example cited in [22], where a company was unable to adopt CASE technology because none of the current generation of CASE-tools evaluated allowed import of the existing models.

The example in this paper uses a simplified version of a well known E-R notation [11]. It is not intended to suggest new or more advanced transformations than those used to demonstrate alternative approaches. However, through the use events the technique is also able to support incremental update of models without having to regenerate a complete schema, a feature important in maintenance scenarios.

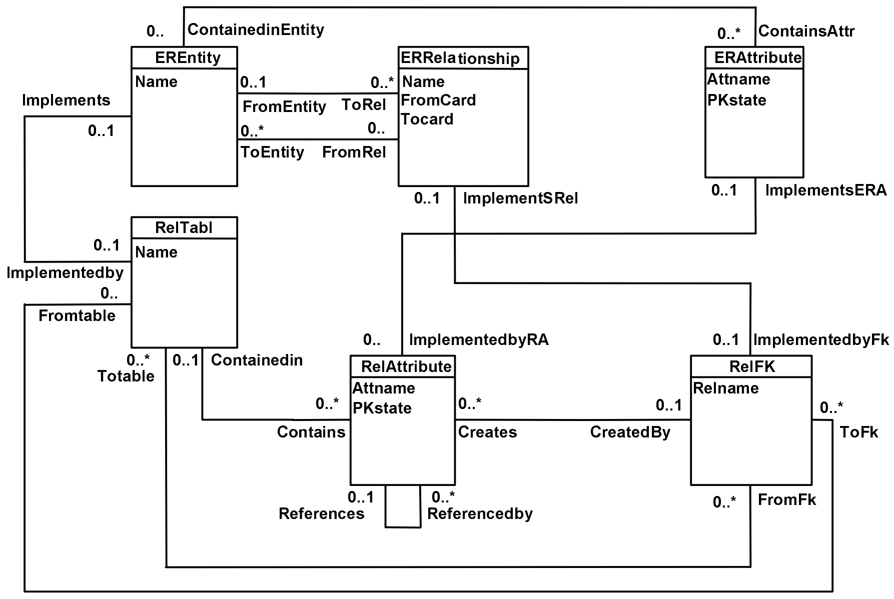

Fig. 1. A simple example meta model without transformation metadata included

The goal in the example is to transform an ER Entity to a relational model table, and transform ER attributes to table attributes while retaining the attribute names and the key attributes of the original entity. This simple transformation requires two separate rules, one that transforms the entities and one that transforms the attributes. An important feature of this rule set is that it can handle incremental updates of the attributes; when a new attribute is added to an existing entity, the corresponding attributes are generated automatically. Since supporting incremental updates of other modelling objects would require more rules, this example only supports incremental updates for the ER attributes. Rules expressed in logic for performing the inverse of this transformation were suggested in [23].

```
Contextclass: EREntity
Event: INSERT
Declaration: RelTable T1
Condition: Implementedby->isempty
Action: T1.create;T1.name:=self.name;
T1.Implements:=self

Contextclass: ERAttribute
Event: INSERT
Declaration: Relattribute RA
Condition: IMPLEMENTEDBYRA->isempty
```

```
Action: RA.CREATE;SELF.implementedbyra:=RA; RA.attname
:=self.attname; RA.containedin :=
self.Containedinentity.implementedby;RA.Pkstate :=
self.PKstate
```

One well-known transformation suggested in another related paper [13] is the transformation of relationships into foreign keys. This transformation is performed by the rule listed below. Another rule would be required to deal with the case when the cardinalities are reversed. Further rules could also be added if options other than generating a foreign key are to be handled.

```
Contextclass: ERRelationship
Event: INSERT
Delcaration: RELFK FK
Condition: Fromcard="1" and Tocard="N" and implemented-
byfk->isempty
Action: FK.create; FK.relname:=self.name;
FK.implementsrel:=self; FK.fromtable :=
self.toentity.implementedby; FK.totable :=
self.fromentity.implementedby

Contextclass: RELATTRIBUTE
Event: IINSERT
Declaration: RELattribute RA
Condition: Referencedby->isempty and containe-
din.fromFK->notempty and PKstate="1"
Action: containedin.fromfk->iterate(
          FK| RA.create; RA.containedin:=fk.fromtable;
          RA.PKstate:="0"; RA.attname:=self.attname;
          RA.createdby:=FK; RA.references:=self)
```

The example rule set also contains rules to perform cascade update and cascade delete of ER attributes. These rules will allow the tool user to delete attributes or to rename ER attribute names without having to regenerate the whole relational model. Using events significantly reduces the number of conditions to be evaluated to support this type of behaviour.

```
Contextclass: ERATTRIBUTE
Event: DELETE
Condition: IMPLEMENTEDBYRA->notempty
Action: IMPLEMENTEDBYRA.REFERENCEDBY->iterate(RA |
RA.delete); IMPLEMENTEDBYRA.DELETE
Contextclass: ERATTRIBUTE
Event: UPDATE ATTNAME
Condition: IMPLEMENTEDBYRA->notempty
Action: IMPLEMENTEDBYRA.REFERENCEDBY->iterate(RA |
RA.attname:=self.attname); IMPLEMENTEDBYRA.attname :=
self.attname
```

4 Analysis and Discussion

We have proposed an approach to concisely representing design transformations through the use of reactive behaviour, implemented using active rules expressed in a conservative[2] extension of the OCL language. The approach allows a compliant tool to use an enhanced set of design transformations for a set of models without the tool itself having to be modified. Since the language used to represent transformations is based on standardized languages and representation techniques (UML/MOF), with standardised ways to interchange models (XMI), existing tools or repository systems can be extended for compliance with relative ease. The approach has advantages over proposals which use proprietary languages, such as [15] and [23]. Examples have been presented to illustrate efficient, incremental update of interchanged models is supported using the approach.

There are a number of implications for tool vendors wishing to benefit from the proposed approach; we perceive three different classes of tool that may utilize an imported model or export a tool-specific model.

Tools in the first class have no support for alternative design transformations, their representation or interchange. Design transformation information has no meaning to these tools, and will be ignored. Such a tool can only usefully export a model; any importing tool must be provided with the transformations used. Building the corresponding model connections would be a non-trivial task.

Tools in the second class allow a user to select from a set of alternative design transformations. For collaboration purposes, these selections should be interchangeable using standardised export formats. Translators must be written for this purpose. Any information that documents a transformation alien to the tool has no meaning to it. The importing translator for the tool has to distinguish the associations between ordinary modelling objects from those between modelling objects and transformation pattern objects, which behave differently. As with class 1, any tool importing from a class 2 tool must be provided with the transformations used.

Tools in the final (class 3) class support all of the different types of information. Such a tool has to distinguish between the various association types only by looking at the meta information. From our earlier analysis, this final class is likely to require support for global navigation between the different modelling objects, which together form a complete set of models for a given domain.

To turn a class 2 tool into a class 3 tool, it is only necessary to develop an extension of the internal transformation engine to interpret OCL actions. The prototype implementation has shown this to be a modest extension to existing OCL interpreters.

[2] In the sense that existing tools can be readily updated to support the extensions.

References

1. Belaunde, M.: A Pragmatic Approach for Building a Flexible UML Model Repository In: UML 1999, Lecture Notes in Computer Science, Vol. 1723. Springer-Verlag, Berlin Heidelberg New York (1999) 188-203.
2. Bernstein, P.A.: Generic Model Management: A Database Infrastructure for Schema Manipulation. In: 9th International Conference on Cooperative Information Systems, LNCS 2172, Springer (2001) 1-6
3. Bernstein, P.A., Bergstraesser, T.: Meta-Data Support for Data Transformations Using Microsoft Repository. IEEE Data Engineering Bulletin 22(1), IEEE (1999) 9-14
4. Blaha, M.R., LaPlant, D., Marvak, E., Requirements for Repository Software. In: WCRE'98 Honolulu, Hawaii, USA. IEEE Computer Society Press (1998) 164-173
5. Brinkkemper, S.: Integrating diagrams in CASE tools through modelling transparency. Information and Software Technology 35(2) (1993) 101-105
6. Dayal, U.: Ten Years of Activity in Active Database Systems: What Have We Accomplished? In: ARTDB 1995, Workshops in Computing, Springer-Verlag, Berlin Heidelberg New York (1995) 3-22
7. Demuth, B, Hußmann, H, Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: UML 2001, Springer-Verlag, Berlin Heidelberg New York (2001) 104-117
8. Gal, A., Etzion, O.: Handling Change Management using Temporal Active Repositories. In: OOER 1995, Lecture Notes in Computer Science, Vol. 1021 Springer-Verlag, Berlin Heidelberg New York (1995) 378-387
9. Gray, J.P., Liu, A., Scott, L.: Issues in software engineering tool construction. Information and Software Technology 42 (2000) 73-77
10. Gustavsson, H., Lings, B.: CASE-tool interchange of Design Transformations. In: 18th British National Conference on Databases: BNCOD 2001, Lecture Notes in Computer Science, Vol. 2097. Springer-Verlag Berlin (2001) 75-88
11. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. 3rd edn. Addison-Wesley, Reading (2000)
12. Fahrner, C., Vossen, G.: A survey of database design transformations based on the Entity-Relationship model. Data & Knowledge Engineering 15(3) (1995) 213-250
13. Hainaut, J.-L.: Specification preservation in schema transformations – application to semantics and statistics. Data & Knowledge Engineering 19 (1996) 99-134
14. Hainaut, J.-L., Henrard, J., Hick, J.M., Roland, D., Englebert, V.: Database Design Recovery. In: CAiSE 1996 Lecture Notes in Computer Science, Vol. 1080. Springer-Verlag, Berlin Heidelberg New York, (1996) 272-300
15. Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.M., Roland D.: Database reverse engineering: From Requirements to CARE tools. Automated Software Engineering 3, 1996, Kluwer Academic Publishers (1996) 9-45
16. Halpin, T.A., Proper, H.A.: Database Schema Transformation & Optimization. In: OOER'95: 14th International Conference on Conceptual Modeling, Springer Lecture Notes in Computer Science, Vol. 1021, Springer-Verlag, Berlin (1995) 191-203
17. Jarke M., Rose T.: Managing Knowledge about Information System Evolution. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD Record 17(3) (1988) 303-311
18. Jasper, H.: Active Databases for Active Repositories. In: ICDE 1994, IEEE Computer Society (1994) 375-384

19. Keller, R., Bédard, J.-F., Saint-Denis, G.: Design and Implementation of a UML-Based Design Repository. In: CAiSE 2001, Lecture Notes in Computer Science 2068, Springer-Verlag, Berlin Heidelberg New York (2001) 448-464
20. Kleppe, A., Warmer, J.: The Object Constraint Language: Precise Modeling with UML, Addison-Wesley (1999)
21. Kleppe, A., Warmer, J.: Extending OCL to include actions. In: 3rd International Conference on the Unified Modelling Language, UML, Springer-Verlag, Berlin Heidelberg New York (2000)
22. Lundell, B., Lings, B., Gustafsson, P.-O.: Method support for developing evaluation frameworks for CASE tool evaluation. In: 1999 Information Resources Management Association International Conference - Track: Computer-Aided Software Engineering Tools, IDEA Group Publishing, Hershey (1999) 350-358
23. McBrien, P., Poulovassilis, A.: A Uniform Approach to Inter-Model Transformations. In: Advanced Information Systems Engineering, 11th International Conference CAiSE'99, Lecture Notes in Computer Science, Vol. 1626, Springer-Verlag, Berlin Heidelberg New York (1999) 333-348
24. Medvidovic, N., Rosenblum, D.S., Redmiles D.F., Robbins, J.E.: Modeling software architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology 11(1) (2002) 2-57
25. Object Management Group: XML Metadata Interchange (XMI) Document ad/98-10-06, http://www.omg.org/docs/ad98-10-05.pdf (1998)
26. Object Management Group: OMG Unified Modeling Language Specification, Version 1.3, June 1999 (1999)
27. Object Management Group: Formal MOF 1.3 Specification formal/00-04-04, http://cgi.omg.org/cgi-bin/doc?formal/00-04-03.pdf (2000)
28. Paton, N.W., Diaz, O.: Active database systems. ACM Computing Surveys 31(1) (1999) 63-103