A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams

Zohaib Khai[1], Aamer Nadeem[1], and Gang-soo Lee[2]

[1] Center for Software Dependability,
Mohammad Ali Jinnah University (MAJU), Islamabad, Pakistan
raja_zohaibkhai@yahoo.com, anadeem@jinnah.edu.pk
[2] Department of Computer Engineering,
Hannam University, Korea
gslee@hannam.ac.kr

Abstract. UML is an industrial standard for designing and developing object-oriented software. It provides a number of notations for modeling different system views, but it still does not have any means of meticulously checking consistency among the models. These models can contain overlapping information which may lead to inconsistencies. If these inconsistencies are not detected and resolved properly at an early stage, they may result in many errors in implementation phase. In this paper, we propose a novel approach for consistency checking of class and sequence diagrams based on Prolog language. In the proposed approach, consistency checking rules as well as UML models are represented in Prolog, then Prolog's reasoning engine is used to automatically find inconsistencies.

Keywords: UML, Sequence Diagram, Class Diagram, Prolog, Constraints, Consistency checking.

1 Introduction

In system development lifecycle, design phase plays an important role as a basis for implementation. During design phase system is modeled in such a way as to bridge the gap between analysis and implementation phase. It is desirable to be able to detect model inconsistencies at an early stage, so that the inconsistencies will not be propagated to code or customer deliverables, such as, documentation [9]. If design phase is modeled properly then process of up-gradation and maintenance of system becomes easy.

An important quality of design is that it should be understandable. To increase the design understandability different design methods and notations have been developed. But for the past few years Unified Modeling Language (UML) [1] is accepted as an industrial standard for object-oriented system modeling. The software design is usually represented as a collection of UML diagrams. UML is a very flexible modeling language as it provides number of notations for modeling different system perspectives, e.g., static view (class diagram) and dynamic view (sequence diagram).

T.-h. Kim et al. (Eds.): ASEA/DRBC/EL 2011, CCIS 257, pp. 85–96, 2011.

It also has a wide range of tools covering up all the features of system modeling for complete and comprehensive representation.

Cost of software development also decreases by performing consistency checking between different UML artifacts. Especially after the emergence of MDA [21] object-oriented code can be generated from UML models. So, for reliable and correct code generation UML artifacts need to be consistent for which model consistency checking is desirable. Similarly modifications in model are relatively simple as compared to the changes in source code. After modifications in model, once again consistency checking is required to validate models.

In this paper we present an approach that transforms UML models into Prolog [3] to perform consistency checking. Use of Prolog is motivated by the fact that it is a declarative programming language that provides beneficial assistance representing arbitrary concepts based on inference rules. It is also quite expressive for the types of consistency rules we deal with.

2 Inconsistency and Consistency Checking Types

This section describes the consistency checking types as given by Mens et al. [12].

2.1 Consistency Checking Types

Vertical Consistency. Consistency checking is performed between diagrams of different versions or abstraction-levels. Syntactic and semantic consistencies are also included in it.

Horizontal Consistency. Consistency checking is performed between different diagrams of same version.

Evolution Consistency. Consistency checking is performed between different versions of a same UML-artifact.

Inconsistencies we consider include both structural, which appears in class diagram, and behavioural that appears in sequence diagram. Classes of inconsistencies used below are taken from [17]. In current paper we deal with the type of consistency known as horizontal consistency. Inconsistencies that can occur are described in next section.

2.2 Inconsistency Types

Dangling Feature Reference [17]. This type of inconsistency occurs when message in sequence diagram references to a method that does not exists in class diagram.

Multiplicity Incompatibility [17]. This type of inconsistency takes place when the link in sequence diagram does not follow the multiplicity constraints defined by corresponding association in class diagram.

Dangling Association Reference [17]. This type of inconsistency occurs when a link is defined between objects in sequence diagram and it has no association between classes of corresponding objects.

Classless Connectable Element [17]. This type of inconsistency occurs when object's lifeline in sequence diagram refers to the class that does not exist in class diagram.

3 Related Work

In this section, the existing UML model consistency checking techniques are discussed. For consistency checking many techniques transform the UML model in some intermediate form, by applying the rules presented in different techniques.

Straeten et al [2, 11] present a technique for consistency detection and resolution using DL (Description logic) [14]. The authors present an inconsistency classification. A UML profile is also developed to support consistency of UML artifacts and then LOOM tool [15] is used for translation of developed profile into DL. Consistency checking is performed for messages, associations and classes but not for constraints and multiplicity.

Krishnan [4] presents an approach for consistency checking based on translation of UML diagrams in state predicates. Only UML behavior diagrams are covered. After translation, PVS (prototype verification system) a theorem prover as well as a model checker is used to perform consistency checking.

Muskens et al [5] present an approach for intra and inter phase consistency checking, which makes use of Partition Algebra using verification rules. Consistency checking is performed by deriving the rules from one view and imposing them on the other view. Consistency checking is performed for associations, messages and constraints.

Egyed [6] introduces a consistency checking technique known as View-Integra. In this technique the diagram to be compared is transformed in such a way that it becomes conceptually close to the diagram with which it is going to compare. Consistency checking is performed between same artifacts, one is original and other one is transformed. Technique is partially automated.

Ehrig et al [7] propose a technique to perform consistency checking between sequence and class diagram based on Attributed Typed Graphs and their transformation. Consistency checking is performed for existence checking (means all classes used in sequence diagram exist in class diagram), visibility checking (visibility of classes, attributes and operations, all should be visible to sequence diagram) and multiplicity checking. Their approach is not supported by any tool support.

Briand et al. [8, 18] propose an approach for change impact analysis based on UML models. This technique is applied before changes are implemented to estimate the effect of change. Some rules are formally defined using OCL to determine the

impact of change on different versions of models. A prototype tool is implemented which also executes consistency checking rules defined.

Paige et al. [10] present an approach to formalize and describe the implementation of consistency constraints between two views of BON (Business Object Notation) i.e. class and collaboration diagram. PVS theorem prover is used to automate the proofs. Consistency checks performed includes sequencing consistency checks (order of message calls), class existence and routine (operation) existence.

Storrle [16] proposes a Prolog based model representation and query interface for analysis of models in MDD (Model Driven Development) setting. Models and queries are represented on the basis of representation defined for Prolog clauses. Queries are used for identifying elements, properties and sub-models of models.

4 Proposed Approach

The proposed approach is an idea of checking consistency of two UML diagrams, i.e., class diagram and sequence diagram. For this purpose, UML models as well as consistency rules are represented in Prolog and then reasoning is performed in Prolog. Our technique provides better coverage of the models and can also be extended to check consistency between OCL constraints of both UML artifacts. Flow diagram of proposed approach is given in Figure 1.

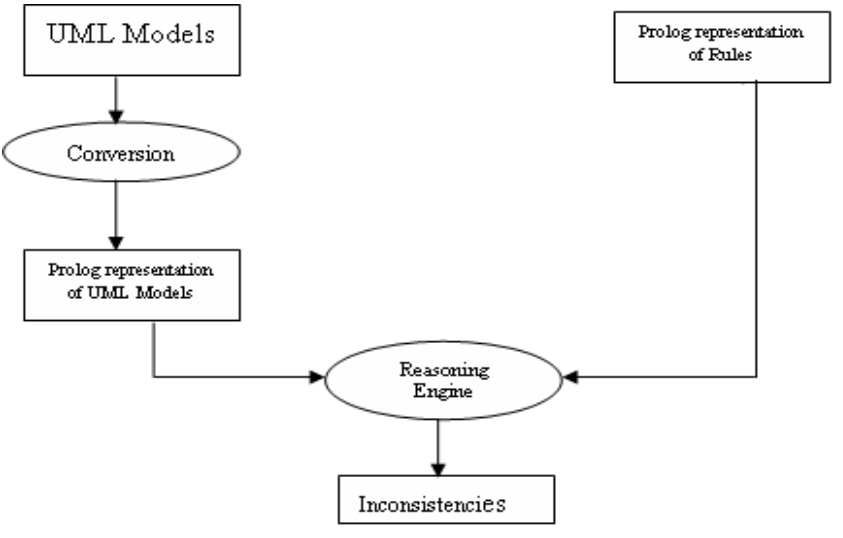

Fig. 1. Flow diagram of Proposed Approach

4.1 Representation of UML Models in Prolog

Class Diagram

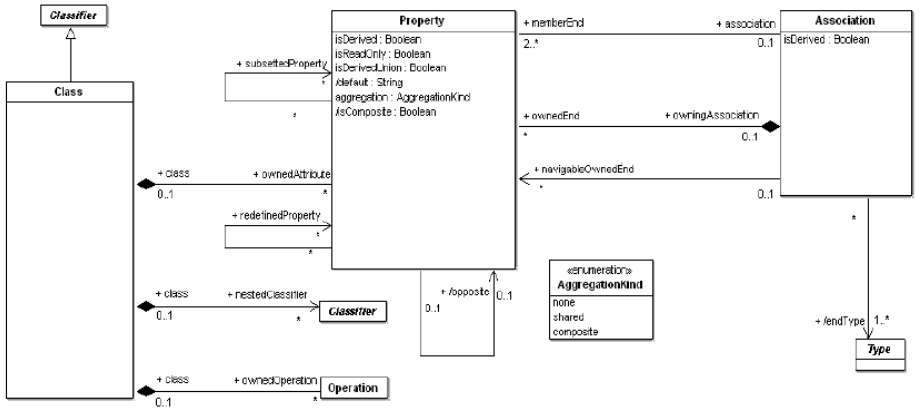

Fig. 2. Partial UML Meta-model for Class diagram [1]

This section deals with Prolog representation of UML class diagram. Figure2 shows partial meta-model of UML class diagram [1]. Class diagram consist of attributes, operations, associations and multiplicity constraints along with class name. Every element of class diagram is assigned an identifier which is a number. These assigned identifiers are used so that relationships between different elements can be created. General format of rules is as follows.

Predname(id(s) , Element(s)-details).

class(Classid , Classname). (1)

'class' used at start is predicate name after that 'Classid' is the unique id assigned to the class and 'Classname' is the actual name of class.

attribute(Attrid , Attrname, Attr-type, Classid). (2)

'attribute' written at the start is predicate name, first thing after brackets is 'Attrid' which is identifier of attribute, second is 'Attrname' i.e. attribute name, third is 'Attr-type' i.e. type of attribute and at fourth place 'Classid' is identifier of class to whom this attribute belongs.

Operation(Opid , Opname , [Parameter(s)-id] , Classid). (3)

'operation' is predicate name, then 'Opid' is operation identifier, 'Opname' is operation name, '[parameters]' is list of parameters and 'Classid' is same as in (1).

parameter(Pid ,Pname, Ptype). (4)

Keyword ‘parameter’ is predicate name, Pid is parameter identifier, ‘Pname’ is name of parameter and ‘Ptype’ refers to the type of parameter, it can either refers to primitive types or to a class.

association(Associd , ClassAid , ClassBid). (5)

Keyword ‘association’ is predicate name, ‘Associd’ is identifier for association. ‘ClassAid’ and ‘ClassBid’ are identifiers for the association ends.

Multiplicity(Associd , Classid , Lowval , Upval). (6)

Keyword ‘multiplicity’ is name of predicate, from ‘Associd’ and ‘Classid’, we come to know the association and class to which multiplicity constraints belongs. ’Lowval’ & ‘Upval’ contains actual values of multiplicity.

Sequence Diagram

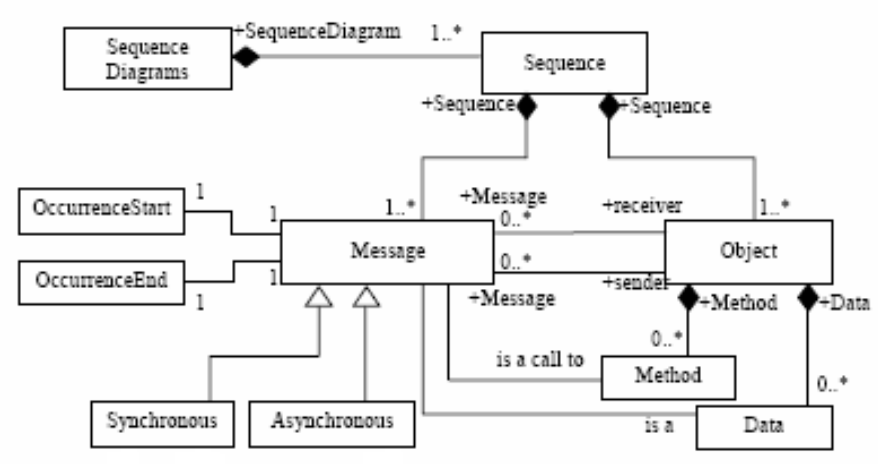

Fig. 3. UML Meta-model for Sequence diagram [19]

This section contains details of Prolog representation of sequence diagram elements. Figure 3 shows meta-model of sequence diagram [19]. Elements of sequence diagram are objects and operation/method call. Similarly identifiers are assigned to different elements of sequence diagram.

object(Objid , Objname, Classid, Multiobj). (7)

Keyword ‘object’ is actually name of predicate. ‘Objid’, ‘Objname’ and ‘Classid’ are object identifier, object name and class identifier respectively. ‘Multiobj’ has value of T(true) or F(false), which tells whether multiple instances exist or not.

mcall(Msgid ,Opname ,[Parameter-list] ,Sndobjid ,Recobjid). (8)

Keyword ‘mcall’ stands for method-call is predicate name. ‘Msgid’, ‘Opname’ and [Parameter-list] are for message identifier, operation name and parameter-list of operation respectively. ‘Sndobjid’ and ‘Recobjid’ is sending object name and receiving object name.

Below is an example of class and sequence diagram representation according to our proposed representation of UML model in Prolog. Both diagrams are taken from [20].

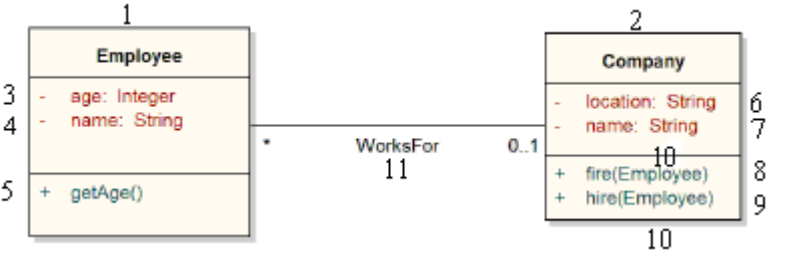

Fig. 4. Example Class Diagram [20]

```
class(1, Employee).
attribute(3, age, integer, 1).
attribute(4, name, String, 1).
operation(5, getAge, [ ], 1).
class(2, Company).
attribute(6, location, String, 2).
attribute(7, name, String, 2).
operation(8, fire,[10], 2).
operation(9, hire, [10], 2).
parameter(10, Employee, Employee).
association(11, 1, 2).
multiplicity(11, 1, 0, 1).
multiplicity(11, 2, 0, n).
```

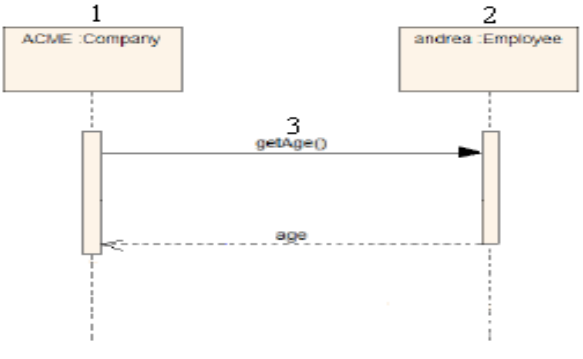

Fig. 5. Example Sequence Diagram [20]

```
object(1, ACME, 2, F).
object(2, Andrea, 1, T).
mcall(3, getAge, [ ], 1, 2).
```

4.2 Consistency Checking Rules

In this section we have proposed some rules for consistency checking of UML models based on Prolog predicate.

Classless Connectable Element. Occurs when object's lifeline in sequence diagram refers to the class that does not exist in class diagram. Here object's information is brought from database using object clause and from that information class identifier is extracted and compared with all class predicates in database to check class existence. In case of negative results of comparison an error message is returned defining inconsistency.

```
objcl_exist_rule(ClCele):-
          object(Objid,_,_,_),
          object(Objid,N,Classid,_),
          ((\+class(Classid,_),
          ClCele="Error: 301");
          fail.
```

Dangling Feature Reference. Occurs when message call in sequence diagram refers to method that does not exist in class diagram. In this rule using method call clause, required information is brought from database and from that information object identifier is extracted to find out the related object. Finally operation existence in corresponding class is checked on the basis of information taken from object. If operation does not exist an error message is returned.

```
op_exist_rule(DfRef):-
          mcall(Msgid,_,_,_),
          mcall(Msgid,Opname,_,Recobjid),
          object(Recobjid,_,Classid,_),
          ((\+operation(_,Opname,_,Classid),
          DfRef="Error: 302");
          fail.
```

Dangling Association Reference. Occurs when there is link between objects in sequence diagram while no association between corresponding classes of class diagram. In this rule first required information about method call and object is gathered from database using 'mcall' and 'object' clauses and then on the basis of gathered information comparison is made to check existence of association between classes. If association does not exist an error message is returned.

```
assoc_exist(DaRef):-
          mcall(Msgid,_,_,_),
          mcall(Msgid,_,Sndobjid,Recobjid),
          object(Sndobjid,_,ClassA,_),
          object(Recobjid,_,ClassB,_),
          ((\+association(_,ClassA,ClassB),
          DaRef="Error: 303");
          fail.
```

Multiplicity Incompatibility. Occurs when multiplicity constraints of both artifacts are not matching. In this rule required information is collected from database using 'mcall', 'object' and 'association' clauses. From gathered information, receiving object is checked whether it's a multi-object or not and on the basis of this further comparison is made to check the multiplicity constraints. If constraints are non-matching then an error message is returned containing details of inconsistency.

```
mlp_in_rule(MulIn):-
          mcall(Msgid,_,_,_),
          mcall(Msgid,_,Sndobjid,Recobjid),
          object(Sndobjid,_,ClassAid,_),
          object(Recobjid,_,ClassBid,BMulti),
          association(Associd,ClassAid,ClassBid),
          ((BMulti == t,
          multiplicity(Associd,ClassBid,_,UpvalB),

          ((UpvalB =< 1,
          MulIn="Error: 304");
          (UpvalB > 1)));

          (BMulti == f,
          multiplicity(Associd,ClassBid,_,UpvalB),

          ((UpvalB < 1,
          MulIn="Error: 304b");
          (UpvalB =:= 1)))),
          fail.
```

5 Automation

Technique proposed in current paper is automatable. For automation of technique certain steps are to be followed. First UML models are converted so that information contained in models can be represented in prolog. This is done by generating XMI of each model, which is by default generated, with each model, in existing CASE tools(e.g. Togeather). Then from XMI relevant information or information to be matched is extracted and represented in the form of Prolog predicates, which are of first order.

After model conversion to prolog predicates, consistency rules from rule database along with converted models are presented to reasoning engine. Reasoning engine performs reasoning on prolog predicates generated from models based on consistency rules and return error code of inconsistencies if any.

6 Evaluation

In this section, evaluation of existing techniques presented in section 3 and our proposed technique is performed. Evaluation is performed on the basis of inconsistency types described in section 2. Result of evaluation is presented below in the form of a table.

Table 1. Comparison of Existing Related Techniques

Inconsistency Types → / Techniques ↓	DFR	MI	DAR	CCE	CI
Simmonds et al (2004), Straeten et al (2003)	Yes	No	Yes	Yes	No
Krishnan, P. (2005)	Yes	No	No	Yes	No
Muskens et al (2005)	Yes	No	Yes	No	Yes
Egyed, A. (2001)	Yes	No	Yes	Yes	No
Ehrig et al (2000)	Yes	Yes(partial)	Yes	Yes	No
Briand et al (2006, 2003)	Yes	No	No	Yes	No
Paige et al (2002)	Yes	No	No	Yes	No
CCSP	Yes	Yes	Yes	Yes	No

Table 2. Abbreviations used in Table1

Abbreviation Used	Value
DFR	Dangling Feature Reference
MI	Multiplicity Incompatibility
DAR	Dangling Association Reference
CCE	Classless Connectable Element
CI	Constraint Incompatibility
CCSP	Consistency checking of Class & Sequence diagram using Prolog

7 Conclusion and Future Work

UML is an industrial standard for designing and developing object-oriented software. To obtain consistent and correct information from UML artifacts, consistency checking of artifacts is required. Also consistency checking plays very important role in reliable and correct code generation in MDD setting, as correct code is generated only if models are consistent. In this paper we present a prolog based consistency checking technique for two different artifacts of UML, Proposed technique provides better diagram coverage and also covers more inconsistency types. Further work can be done by including more elements of both artifacts. More artifacts can also be added by covering all elements of those artifacts to avoid skipping minor details in models.

Ackmowledgement. This work was supported by the Security Engineering Research Center, under research grant from the Korean Ministry of Knowledge Economy.

References

1. Object Management Group. Unified Modeling Language specification version 2.1.2. formal/2007-11-01 (November 2007)
2. Simmonds, J., Van Der Straeten, R., Jonckers, V., Mens, T.: Maintaining consistency between UML models using description logic. In: Proceedings Langages et Modèles à Objets 2004, RSTI série L'Objet, vol. 10(2-3), pp. 231–244. Hermes Science Publications (2004)
3. Clocksin, W.F., Mellish, C.S.: Programming in Prolog, 2nd edn. Springer, Heidelberg (1984)
4. Krishnan, P.: Consistency Checks for UML. In: The Proc. of the Asia Pacific Software engineering Conference (APSEC 2000), pp. 162–169 (December 2000)
5. Muskens, J., Brill, R.J.: Generalizing Consistency Checking between Software Views. In: Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 169–180 (2005)
6. Egyed, A.: Scalable consistency checking between diagrams -The VIEWINTEGRA Approach. In: Proceedings of the 16th International Conference on Automated Software Engineering, San Diego, USA (November 2001)
7. Ehrig, H., Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams using Attributed Typed Graph Grammars. In: Proceedings of joint APPLIGRAPH/ GETGRATS workshop on Graph Transformation systems, Berlin (March 2000)
8. Briand, L.C., Labiche, Y., O'Sullivan, L., Sowka, M.M.: Automated Impact Analysis of UML Models. Journal of Systems and Software 79(3), 339–352 (2006)
9. Paige, R.F., Ostroff, J.S., Brooke, P.J.: A Test-Based Agile Approach to Checking the Consistency of Class and Collaboration Diagrams, UK Software Testing Workshop, University of York, September 4-5 (2003)
10. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Checking the Consistency of Collaboration and class Diagrams using PVS. In: Proc. Fourth Workshop on Rigorous Object-Oriented Methods. British Computer Society, London (March 2002)
11. Straeten, R.V.D., Mens, T., Simmonds, J.: Maintaining Consistency between UML Models with Description Logic Tools. In: ECOOP Workshop on Object-Oriented Reengineering, Darmstadt, Germany (July 2003)
12. Mens, T., Straeten, R.V.D., Simmonds, J.: A Framework for Managing Consistency of Evolving UML Models. In: Yang, H. (ed.) Software Evolution with UML and XML, ch.1. Idea Group Inc. (March 2005)
13. Usman, M., Nadeem, A., Tai-hoon, K., Cho, E.-S.: A Survey of Consistency Checking Techniques for UML Models. Advanced Software Engineering and Its Applications, pp. 57–62. ASEA, Hainan Island (2008)
14. Baader, F., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
15. MacGregor, R.M.: Inside the LOOM description classifier. ACM SIGART Bulletin 2(3), 88–92 (1991)
16. Störrle, H.: A PROLOG-based Approach to Representing and Querying UML Models. In: Cox, P., Fish, A., Howse, J. (eds.) Intl. Ws. Visual Languages and Logic (VLL 2007). CEUR-WS, vol. 274, pp. 71–84. CEUR (2007)
17. Straeten, R.V.D.: Inconsistency management in model-driven engineering using description logics. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium (September 2005)

18. Briand, L.C., Labiche, Y., O'Sullivan, L.: Impact Analysis and Change Management of UML Models. In: Proceedings of the 19th International Conference Software Maintenance (ICSM 2003), pp. 256–265. IEEE Computer Society Press, Amsterdam (2003)
19. Ouardani, A., Esteban, P., Paludetto, M., Pascal, J.: A Meta-modeling Approach for Sequence Diagrams to Petri Nets Transformation within the requirement validation process. In: The 20th annual European Simulation and Modeling Conference, ESM 2006 conference, LAAS, Toulouse, France (2006)
20. Baruzzo, A.: A Unified Framework for Automated UML Model Analysis. PhD thesis, Department of Mathematics and Computer Science, University of Udine, Italy (July 2006)
21. Object Management Group (OMG), MDA Guide, Version 1.0.1 (2003), http://www.omg.org/docs/omg/03-06-01.pdf