A Constraint-based Approach for Checking Vertical Inconsistencies between Class and Sequence UML Diagrams

Driss Allaki, Mohamed Dahchour and Abdeslam En-Nouaary
Institut National des Postes et Télécommunications, 2, av ALLal EL Fassi – Madinat AL Irfane, Rabat, Morocco

Keywords: UML, Vertical Inconsistencies, Software Development Process, Constraints, Metamodel.

Abstract: The modern software development processes enable evolving software systems and refining models across software life cycle. However, these evolution attitudes may lead to some consistency problems among models at different levels of abstraction. Hence, it is required to discover and detect the potential inconsistencies occurring in models when developing a system. This paper focuses on checking the vertical consistency of UML models using an approach based on defining constraints at the meta-level. These constraints are expressed using EVL (Epsilon Validation Language) to ensure the consistency of models. Representative examples of constraints for checking vertical inconsistencies between class and sequence diagrams are proposed to illustrate our contribution.

1 INTRODUCTION

Over the past few years, modeling systems has long been an essential practice in software development, since a model is supposed to anticipate the results of coding. Indeed, a model is an abstract representation of a system intended for understanding, studying and documenting the system (Cernosek and Naiburg, 2004). Each member of the project team, from the user to the developer, uses and enriches the model differently. Also, the model has the particular advantage of facilitating traceability of the system, namely the possibility of starting from one of its components and monitors its interactions and relationship with other parts of the model.

To illustrate what a model is, *Grady Booch* draws a parallel between a software development and a building construction. This analogy is appropriate since the plots plans to construct a building perfectly reflects the idea of anticipation, design and documentation of the model. However, we note that in building modeling, this anticipation does not take into account the changing needs of users, the starting hypothesis is that these needs are defined once and for all. Yet, in many cases, in software development, these needs change over the project; that is why it is important to manage change and recognize the need to continue supporting our models. Then, unlike what is done in the construction industry, the software modeling process must be adaptive rather than predictive.

From this perspective, a software modeling process defines a sequence of steps, partially ordered, which contribute to the realization of software or changing an existing system (Jacobson et al., 1999). Then, the purpose of a development process is to produce quality software that meets the changing needs of the users in predictable time and cost. To this end, most of modern software modeling processes adopt *iterative* and *incremental* strategies as is the case in *agile* context. The iterative approach is based on the growth and the successive refinement of a system through multiple iterations, feedback and cyclical adjustment being the main engines to converge on a satisfactory system. In the *incremental* development, we split the tasks into small parts, plan them to be developed over time and incorporate them as soon as they are completed. When *agile* modeling is based on some simple principles with common sense that encourage changing models perspectives if needed, and motivate creating multiple models simultaneously.

According to (Huzar et al., 2004), the incremental and iterative nature of software systems and the agile and flexible software modeling processes are one of the main causes of model inconsistencies. An inconsistency roughly means that overlapping elements of different model aspects do not match each other (Allaki et al., 2014). Or in other words, the whole system is not represented in an harmonized way in different views of its model.

These inconsistencies could be the source of many errors and could therefore invalidate the models and complicate the whole software development process. Especially when adopting a Model Driven Engineering (MDE) approach (Schmidt, 2006). The Object Management Group vision of MDE is called Model Driven Architecture (MDA, 2003). MDA formulates well-established rules and good practices such as adopting the Unified Modeling Language (UML, 2015) as a *de facto* standard for modeling software systems. UML is defined as a graphical and textual modeling language composed of multiple diagrams that unifies both notations and object-oriented concepts. The concepts transmitted by a diagram have a precise semantics and are carriers of meaning. For example, semantics expressed by class and sequence diagrams makes them the most complementarily related diagrams containing meaningful information about both the structure and the behavior of the system being investigated; which makes them also the most refined diagrams during all different software development phases. For this reason, we consider and focus in this work, on examples of inconsistencies between these two diagrams.

In this paper, we first explain, using examples, *how scalable development processes using iterative, incremental and adaptive methods are behind the occurrence of vertical (inter-model) inconsistencies* (i.e. inconsistencies arising among UML model diagrams at different levels of abstraction). After that, we will describe *how our proposed constraint-based consistency checking proposal works*, and we will propose, thereafter, a set of constraints that deal with the given examples of vertical inconsistencies between class and sequence diagrams introduced before.

The rest of the paper is organized as follows. Section II provides three motivating examples of vertical inconsistencies between class and sequence diagrams. Section III presents our constraint-based approach for checking UML model inconsistencies, illustrated by examples dealing with the given vertical inconsistencies, and discussed according to the advantages and limitations of related works.

2 VERTICAL INCONSISTENCIES BETWEEN CLASS AND SEQUENCE DIAGRAMS

The inherent complexity of software systems during their creation will continue to grow as they are evolving, either using traditional or agile software development processes. Indeed, mixing between iterative, incremental and adaptive strategies affects models' consistency by adopting some change attitudes in different development phases. More explicitly, these attitudes advocate assuming models' simplicity, enabling change, using multiple models and so on. The cited attitudes encourage to not over-modeling the system in the first steps of development; which means not depicting additional features in our models until the system requirements evolve in the future. This can be done by developing a small model, or perhaps a high-level model, and evolve it over time (or simply discard it when no longer need it) in an incremental manner. Moreover, we have to use multiple models to develop software, depending on the exact nature of the software we are developing. All these attitudes can lead to numerous conflicts in models across different levels of abstraction. Thus, vertical inconsistencies can arise as a result.

Being aware of this fact, particular attention should concern checking this kind of inconsistencies, as well as others, to undergo changes during a software life cycle, correct errors, accommodate new requirements, and so on.

In what follows, we present some motivating examples from literature that illustrate the conflicts arising between class and sequence diagrams at different levels of abstraction.

Hereafter, we consider that the different parts of the sequence diagrams presented in the following examples are a refinement, at the instance level, of an existing sequence diagram defined in a higher level (specification level). The refinement is used to present more details on the interaction between the objects used in these examples. This lead to assume that the class diagrams are on a higher level of abstraction than the given sequence diagrams.

Example 1: (Connector Type Incompatibility)

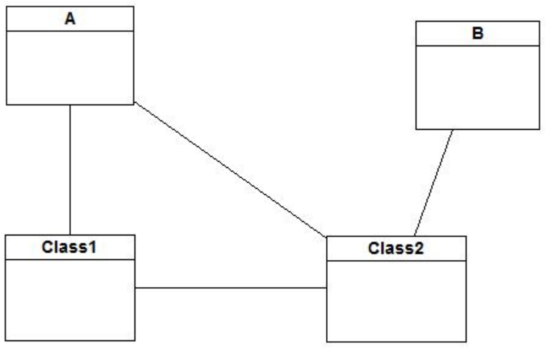

Figure 1: A part of a class diagram (1).

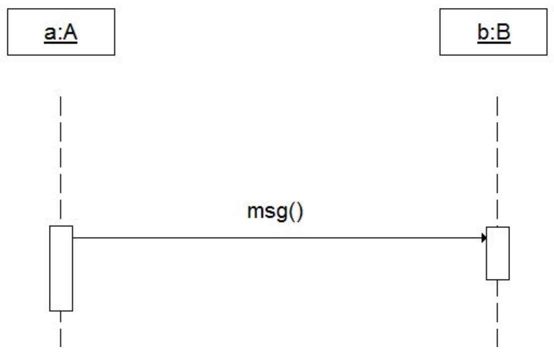

Figure 2: A part of a sequence diagram (1).

The part of sequence diagram illustrated in figure 2 shows an instance of class "A" sending a new introduced message "msg" to an instance of class "B" although there is no direct relationship between the two classes "A" and "B" in the class diagram of figure 1.

If we consider, for example, an incremental context, this inconsistency could occur when we are developing, in the phase of design, a new increment of the software system. For instance, when adding new functionalities to the system, in this new under development increment, we can introduce a new message that links between two objects of two existing classes without updating the class diagram by linking these two classes. Or without editing the sequence diagram in progress to be sure that all messages link only between related classes. This kind of attitudes is common and may appear in an unnoticed way in the context of refining the design of the system being developed following an incremental strategy.

Example 2: (Dangling Operation)

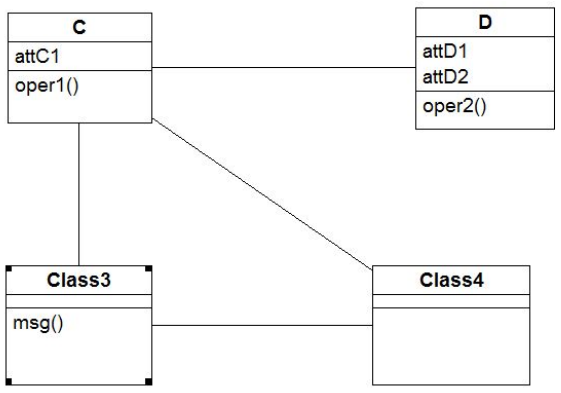

Figure 3: A part of a class diagram (2).

The part of sequence diagram illustrated in figure 4 represents an instance of class "C" sending a new introduced message "msg" to an instance of class "D". However, the message "msg" refers to an operation in class diagram illustrated by figure 3 that does not belong to the class "D" attached to the receiving event of the message in sequence diagram.

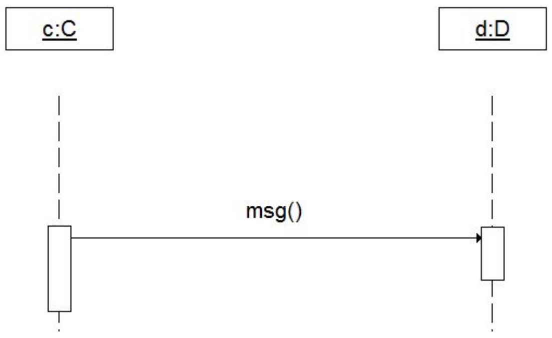

Figure 4: A part of a sequence diagram (2).

When adapting models, for example in an agile software development process, it is common to change design, or part of it, due to the change of initial requirements. This change may lead to some inconsistencies that concern the behavioral aspect of the model. For instance, during these design changes, some operations in the class diagram may not be moved to another class, or sometimes may not be removed from the model. And then, these operations can be referred in a wrong way in the other diagrams; like the case of the *dangling operation* presented before.

Example 3: (Navigation Incompatibility)

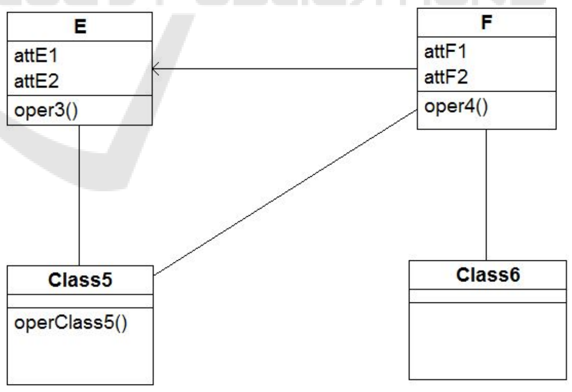

Figure 5: A part of a class diagram (3).

In the part of sequence diagram illustrated in figure 6, a message is sent from a sender object "E" to a receiver object "F" in opposition to the navigation direction of the association between the two corresponding classes "E" and "F" in the class diagram represented in figure 5.

If rearrangements are carried out on an existing part of the system, the example of *navigation incompatibility*

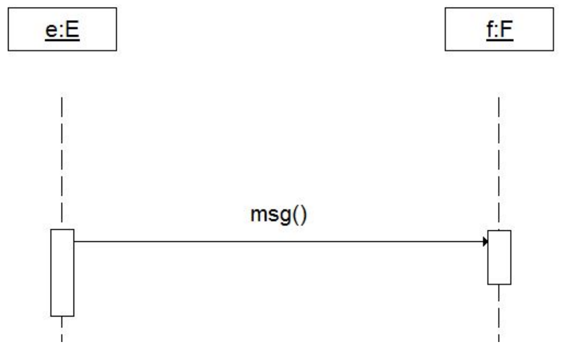

Figure 6: A part of a sequence diagram (3).

could occur. For example, when adopting an iterative strategy in the development process, we develop the least possible before the system is submitted to evaluation. And then, we can neglect, in the first iterations, some details in design; such as the navigation direction of the association between classes. But when refining the model, such information could be added, and then it becomes crucial to adopt the other parts of the model to these changes. Then, for instance, sending a message between two objects without taking into consideration the navigation direction of the association linking between their respective classes is not allowed.

As pointed before, different types of inconsistencies can be encountered in UML models. In this paper, we focus on vertical inconsistencies. The taxonomy presented in (Allaki et al., 2015) proposes more examples and more details about a comprehensive classification of inconsistencies.

3 OUR PROPOSED CONSTRAINT BASED APPROACH

In this section, we present the approach we used for checking the consistency of UML models. Our technique is based on formal constraints defined at the metamodel of UML. These constraints are implemented using EVL (Epsilon Validation Language, 2015) by matching related diagrams' features at the metamodel level.

3.1 An Overview of our Approach

Our EVL constraint-based approach matches UML meta-elements to ensure models' consistency. In our context, the constraints added at the meta-level describe different conditions that UML models have to satisfy to be considered consistent. These conditions concern, syntactically and semantically, the homogeneity, the complementarity and the compatibility of the UML diagrams' elements. Then, checking inconsistencies will be based on detecting violations of consistency according to these constraints. Since the consistency constraints are defined at the UML metamodel level, they have the advantage of being independent from any specific implementation platform and so they can be applied generically to all UML models since any UML model inherits all the specifications, including constraints, from its metamodel.

Note that these constraints will be enabled once the modeler explicitly asks the validation of his model and not during modeling. Thus, some "fake inconsistencies" such as incompleteness or anomalies that are intentionally produced when the model is under construction, could be avoided.

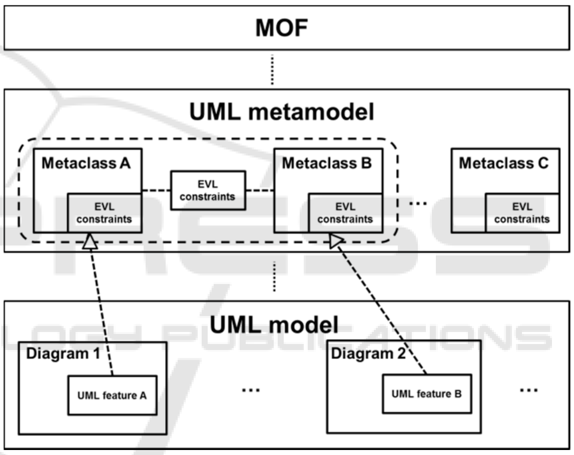

Figure 7: Constraints in UML metamodel level.

On the other hand, recall that UML design models are typically expressed as a large collection of interdependent and partially overlapping UML diagrams. These diagrams relate to different aspects of the system, and are somehow related to each other, as some of their elements have matching links. These links are expressed by the different meta-associations between meta-classes in the UML metamodel.

Our solution exploits these facts to check inconsistencies that can arise between multiple views of the model, even if they are at different levels of abstraction. The key idea behind our approach is a matching between meta-classes by establishing the right links when defining a consistency constraint at UML metamodel. The definition of such constraints is basically done by first, choosing the right meta-classes involved in the constraint, and then, by determining the way these meta-classes are linked.

3.2 Examples of EVL Constraints

In what follows, we produce, for each given example in (Section 2), the UML meta-classes concerned by the inconsistency and the associated constraint expressed in EVL.

EVL (Epsilon Validation Language) is a task-specific language of the general model management language Epsilon (Epsilon, 2015). EVL is a language dedicated to validate models. In their simplest form, constraints expressed in EVL are quite similar to OCL constraints. However, unlike OCL, EVL supports dependencies between constraints (e.g. if constraint A fails, do not evaluate constraint B), supports user interaction (specifies customizable error messages and quick fixes for failed constraints), supports all the usual programming constructs and the convenient first-order logic OCL operations and so on (Kolovos et al., 2015).

All EVL features are suitably integrated in Eclipse Modeling, the CASE tool we used to implement our approach.

Example 1: (Connector Type Incompatibility)
The involved inconsistency elements from the UML metamodel are shown in the following figure.

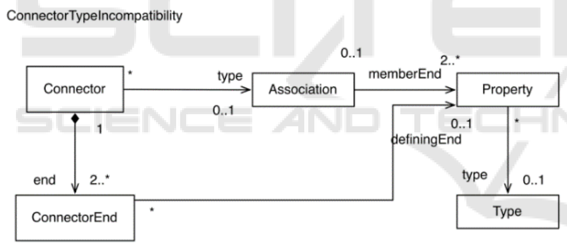

Figure 8: Involved elements from the UML metamodel in the Connector Type Incompatibility inconsistency.

The EVL constraint that checks this inconsistency is presented as follows:

```
    context Connector {

    constraint ConnectorTypeIncompatibility {

    check    :    self.type.memberEnd.type    =
self.end.definingEnd.type

     message  : "A model contains a connector"
+  self.name  +  "for  which  the  type  of  the
connectable elements that are attached to the
ends of the connector don't conform to the type
of the association ends of the association that
types the connector"
    }
    }
```

In this example, we choose the meta-class *Connector* as a context of the EVL constraint. We make sure if the types of the connectable elements that the ends of the connector are attached conform to the types of the association ends of the association that types the connector. And if this inconsistency appears, a message explaining the situation is displayed.

Example 2: (Dangling Operation)
The part of UML metamodel containing the adequate meta-classes involved in this inconsistency is shown in figure 9.

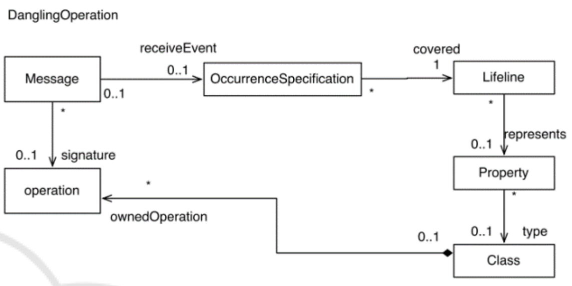

Figure 9: Involved elements from the UML metamodel in the Dangling Operation inconsistency.

In this example, we consider for clarity reasons the simplest instance of the *Dangling Operation* in which we have just one operation in the class. The EVL constraint that checks and fixes this inconsistency is presented as follows:

```
    context Message {

    constraint DanglingOperation{

    check      :      self.signature      =
self.receivedEvent.covered.represents.type.Ow
nedOperation.signature

     message  : "A sequence diagram contains a
message" + self.name + " which refers to an
operation that does not belong to the class
attached to the receiving event of the message"

    fix { title : "add an operation to the
class"
           do { var op = new Operation;
                op.name=
self.name;Class.ownedOperation.first().conten
ts.add(op);
           }
            }
    }
    }
```

To deal with the *Dangling Operation* inconsistency, we choose for the corresponding constraint, the meta-class *Message as a* context. The objective then is to compare the *signature* of the *Operation* referenced by the *Message* with the *signature* of the *Operation* belonging to the *Class* attached to the receiving event of the *Message* in the Sequence diagram. If the two signatures are different, the inconsistency occurs and therefore a useful message is displayed with a proposition of fixing the inconsistency by creating a new operation to the corresponding class.

Example 3: (Navigation Incompatibility)
The involved meta-classes of this inconsistency are shown in figure 10.

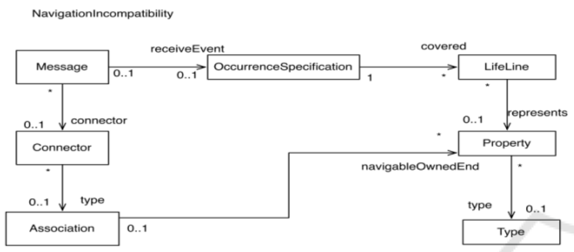

Figure 10: Involved elements from the UML metamodel in the Navigation Incompatibility inconsistency.

The EVL constraint that checks the simplest form of this inconsistency is presented as follows:

```
   context Message {
   constraint NavigationIncompatibility{
   check :
   self.receivedEvent.covered.represents.type
= self.connector.type.navigableOwnedEnd.type
    message : "A sequence diagram contains a
message" + self.name + "of which calling
direction does not match the navigation
constraint on the corresponding association"
   }
   }
```

The context chosen for the *Navigation Incompatibility* constraint is the meta-class *Message*. By defining this constraint, we aim to compare the calling direction of the message if it matches the navigation constraint on the corresponding association. An explanatory message is displayed if the inconsistency arises.

3.3 Discussion

Over the past few years, ensuring consistency in UML models has been a priority investigation for researchers and practitioners in software engineering. As a result, several approaches have been devised to deal with this issue. These approaches can be classified into two categories, namely *transformation-based* techniques and *constraint-based* techniques.
Transformation-based techniques, for example but not limited to (Hanzala and Porres, 2015); (Miloudi et al., 2011), (Straeten et al., 2007) and (Yao and Shatz, 2006) are founded on detecting inconsistencies, after transforming semi-formal UML models to a formal language, using inference mechanisms of that language.

These methods provide us with solid mathematical foundation, proof and tools and add more precision to UML models by avoiding ambiguities when handling inconsistencies in these models.

On the other side, constraint-based techniques, such as (Przigoda et al., 2016), (Kalibatiene et al., 2013), (Sapna and Mohanty, 2007), (Egyed, 2007) and so on, detect inconsistencies in accordance to the formal constraints defined at the metamodel level.

These methods are extensible, by giving the possibility to include new checks for new arising inconsistencies. Also, unlike transformation techniques, they preserve all the information expressed in the UML models; and make the model more expressive through the constraints defined at the metamodel.

However, most of the existing constraint-based proposals generally deal with static aspects of the UML models and are limited to checking inconsistencies in a single diagram, which compromise their efficiency.

Giving the pros and cons of the existing inconsistency checking methods, our proposed constraint-based solution overcomes some of these limitations since it is conceived to ensure the quality and the usefulness of the proposal. Our proposal is easily automated (implemented using Eclipse Modeling). Moreover, EVL, the language used to write constraints, provides much helpful functionality such as the support of quick fixes and the customizable error messages. This can motivate industrial development communities to use it, unlike most of the existing formal techniques that are hard to automate and require a strong mathematical background to apply them. Furthermore, the constraint-based nature of our proposal supports extension mechanisms to deal with any new arising inconsistency. In addition to that, our proposal was designed to be complete in terms of coverage of both potential inconsistencies and the UML diagrams commonly used such as the class, sequence, activity, statechart diagrams and so on; which make it a simple and practical consistency checking proposal.

4 CONCLUSIONS

We tried through this paper to deal with the case of the vertical inconsistencies caused by the refinement of the model. Models are generally refined because of the iterative, incremental and adaptive nature of the modern software development processes.

We explained how our constraint-based consistency checking proposal treats this type of inconsistencies. Our approach adds constraints at the metamodel level by matching the common concepts among the UML diagrams. These constraints, written using the Epsilon Validation Language, automatically help detecting and fixing inconsistencies. To illustrate our approach, we have considered examples of constraints that check vertical inconsistencies arising between class and sequence diagrams.

On the other hand, our proposal is characterized by its ease of automation (implemented using Eclipse Modeling), ability to be extended and completeness of covering all the potential inconsistencies that can affect all the commonly used UML diagrams.

As a future work, we intend to develop a consistency checking process that regroups the best-practices of detecting and handling UML model inconsistencies and that focuses on defining the different steps needed to well behave with the detected inconsistencies. We will apply this on a case study that contains patterns involving a set of tricky examples of inconsistencies and that covers a larger number of expressive UML diagrams. We will also provide further discussion about the experimental results with the Eclipse tool and its performance.

REFERENCES

Cernosek, G., Naiburg, E., 2004. The Value of Modeling.A technical discussion of software modeling. (IBM).

Jacobson, I., Booch, G., Rumbaugh, J., 1999. *Software Development Process*, An Imprint of Addison Wesley Longman, Inc.

Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L., 2004. Consistency problems in UML based software development. In UML Modeling Languages and Applications, «UML» 2004 Satellite Activities, Lisbon, Portugal, October 11-15, 2004, Revised Selected Papers. LNCS, vol. 3297, pp. 1-12.

Allaki, D., Dahchour, M., En-nouaary, A, 2014. A New Taxonomy of Inconsistencies in UML Models: Towards Better MDE. In the Proceedings of the 9th International Conference on Intelligent Systems: Theories and Applications, (SITA'14), May 2014, Rabat, Morocco, pp.121-127.

Allaki, D., Dahchour, M., En-nouaary, A, 2015. A New Taxonomy of Inconsistencies in UML Models with their Detection Methods for better MDE. In *International Journal of Computer Science and Applications, Technomathematics Research Foundation*, Vol.12, No.1, pp.48–65.

Schmidt, D, 2006. Guest editor's introduction: Model-Driven Engineering. In IEEE Computer Society, February 2006, Volume 39, No. 2, pp. 25-31.

MDA Guide Version 1.0.1, <http://www.omg.org/mda>, 2003. (Last accessed November 2015).

Unified Modeling Language: Superstructure. Version 2.5, <http://www.omg.org/spec/UML/2.5/>, 2015. (Last accessed November 2015).

Epsilon Validation Language, 2015. <http://www.eclipse.org/epsilon/doc/evl/>, (Last accessed November 2015).

Epsilon, 2015. <http://www.eclipse.org/epsilon/doc/>, (Last accessed November 2015).

Kolovos, D., Rose, L., Domínguez, A.G., Paige, R., 2015. *The epsilon book*. February 4, 2015.

Hanzala, A. K., Porres, I., 2015. Consistency of UML class, object and statechart diagrams using Ontology Reasoners. In *Journal of Visual Languages & Computing.*Volume 26, February 2015, pp. 42–65.

Miloudi, K. E., Amrani, Y. E., Ettouhami, A. 2011. An Automated Translation of UML Class Diagrams into a Formal Specification to Detect UML Inconsistencies. In The Sixth International Conference on Software Engineering Advances, ICSEA 2011, Barcelona, Spain, pp. 432–438.

Straeten, R.V. D., Jonckers, V., Mens, T. 2007.A Formal Approach to Model refactoring and Model refinement. In *Software and System Modeling*, Volume 6, Number 2, June 2007, pp. 139–162.

Yao, S., Shatz, S. M., 2006. Consistency Checking of UML dynamic models based on Petri Net techniques. In 15th International Conference on Computing (CIC 2006), November 21-24, 2006, Mexico City, Mexico, pp. 289–297.

Przigoda, N., Wille, R., Drechsler, R., 2016. Analyzing Inconsistencies in UML/OCL Models. In *Journal of Circuits, Systems and Computers*, Volume 25, Issue 03, March 2016.

Kalibatiene, D., Vasilecas, O., Dubauskaite, R., 2013.Ensuring Consistency in Different IS Models – UML Case Study. In *Baltic Journal of Modern Computing, Volume 1*, No. 1-2, 2013, pp. 63-76.

Sapna, P. G., Mohanty, H., 2007. Ensuring consistency in relational repository of UML models. In 10th International Conference in Information Technology, ICIT 2007, Roukela, India, 17-20 December 2007, pp. 217–222.

Egyed, A., 2007. Fixing inconsistencies in UML design models. In 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pp. 292-301.