International Journal of Computer Science and Applications,
©Technomathematics Research Foundation
Vol. 12, No. 1, pp. 48 – 65, 2015

A NEW TAXONOMY OF INCONSISTENCIES IN UML MODELS WITH THEIR DETECTION METHODS FOR BETTER MDE

DRISS ALLAKI

Research Laboratory STRS, National Institute of Posts and Telecommunications,
2 Allal EL Fassi avenue Madinat AL Irfane, Rabat, Morocco
d.allaki@inpt.ac.ma

MOHAMED DAHCHOUR

Research Laboratory STRS, National Institute of Posts and Telecommunications,
2 Allal EL Fassi avenue Madinat AL Irfane, Rabat, Morocco
dahchour@inpt.ac.ma

ABDESLAM EN-NOUAARY

Research Laboratory STRS, National Institute of Posts and Telecommunications,
2 Allal EL Fassi avenue Madinat AL Irfane, Rabat, Morocco
abdeslam@inpt.ac.ma

MDE (Model Driven Engineering) is an emerging software engineering paradigm that relies on models as primary artifacts to build complex software and hardware systems. It basically aims at overcoming the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively. One particular vision of MDE was proposed by OMG (Object Management Group) under the name of MDA (Model Driven Architecture). MDA consists of a set of standardized rules and practices to promote good modeling and fully exploit the models in order to gain in sustainability, consideration of execution platforms and productivity. To achieve these objectives, the models to be used within an MDA approach should be consistent (i.e., complete, free of ambiguities, free of contradictions and anomalies, etc.). In this paper, we identify and describe possible inconsistencies that can be encountered in UML models while applying MDA, propose a new taxonomy for such inconsistencies and precise those supported by the most known verification approaches found in literature. Simple examples are used throughout the paper to illustrate our contribution.

Keywords: MDE; MDA ; UML; Model inconsistencies; Formal techniques; Non-Formal techniques.

1. Introduction

Nowadays, software production is more and more facing exponential increase in the complexity of the systems being built, and urgent reduction of development cost and time to market. To cope with these constraints, Model-Driven Engineering (MDE) has emerged as a new paradigm, which formulates well-established rules and good practices agreed on by software engineering researchers and practitioners. The definitive objective of MDE is to rely on models as primary artifacts to overcome the inability of third-

generation languages to alleviate the complexity of platforms and express domain concepts effectively [Schmidt (2006)]. A model is commonly defined as an abstraction or simplification of a system that is necessary and sufficient to understand the system being modeled and to help answer questions about it.

To make models operational (for code generation, validation, verification, implementation, etc.), model transformation mechanisms should be used. A model transformation is a set of rules / correspondences between the elements of the source model and those of the target model. The source and target models may conform to different metamodels (exogenous transformation) or conform to a common metamodel (endogenous transformation).

To promote good modeling practices and fully exploit the advantages of models, the Object Management Group (OMG) proposed as early as 2000 its own vision of MDE, under the name of MDA (Model Driven Architecture) [MDA (2003)]. This approach is based on several standards such as UML [UML (2011)], MOF [MOF (2011)] and XMI [XMI (2013)]. The ultimate goal is to develop UML models across the phases of software development life cycle. To this end, MDA recommends the development of:

- Requirement model (Computation Independent Model - CIM) in which no computer consideration is taken into account;
- Analysis and design model (Platform Independent Model - PIM) in which no specific platform is considered for implementation or validation;
- Code model (Platform Specific Model - PSM) platform-specific. The transition from PIM to PSM involves mechanisms processing model and a description of the platform (Platform Description Model - PDM).

To fully benefit from the advantages of adopting an MDA approach, the models used along the software process should be consistent or free of inconsistencies. An *inconsistency* roughly means that overlapping elements of different model aspects don't match each other. In other words, the whole system is not represented in a harmonized way in different views of its model. It should be noted that several inconsistencies can unfortunately arise when modeling a system. According to [Huzar *et al.* (2004)], these inconsistencies may come from three basic sources:

- The nature of multi-view software systems: A software system is represented as a set of views and models describing the different aspects according to which the system is discussed. These models can overlap and sometimes include conflicting specifications.
- The incremental and iterative nature of software systems: A software system is usually developed through different phases and iterations, and in each phase / iteration we produce new thinner models than previous ones.
- The distributed development nature of software systems: A software system is generally built by a team whose members can be simultaneously involved in the building process.

These inconsistencies could be the source of many errors and could therefore invalidate the MDA models and complicate the process of model transformations. In order to tackle these inconsistencies, we need to identify and describe each of them before trying to correct them.

The interest of identifying, studying and classifying inconsistencies lies first in the fact that we have to reduce the complexity of the problem for a better understanding. And then, to find more ease in resolving separately these inconsistencies which are generally unpredictable, uncountable and can take several aspects.

In this paper[a], we identify and describe possible inconsistencies that can be encountered in UML models while applying MDA, and propose a new taxonomy for such inconsistencies. The classification is based on two combined criteria, namely *terminological* and *typological*. After that, we precise which inconsistencies, among those figuring in our taxonomy, are supported by the most known verification approaches found in literature.

The rest of the paper is organized as follows. Section 2 presents an overview of two well-known classifications of inconsistencies in UML models. Section 3 describes a new taxonomy to classify UML model inconsistencies based on two kinds of classifications called respectively *terminological* and *typological*. Section 4 gives some representative examples to illustrate the various combinations of inconsistencies. Section 5 presents the different approaches used to fix the problem of inconsistencies and precise those supported by these approaches according to our classification. Section 6 concludes the paper and presents future work.

2. An overview of existing classifications of inconsistencies

Over the past few years, several works have been devised for checking inconsistencies in UML models ([Lucas *et al.* (2009)], [Dubauskaite and Vasilecas (2013)], [Noraini *et al.* (2011)], etc.). However, few are devoted to describe, identify, name and classify these inconsistencies. We present in the following an overview of the most important inconsistency classifications encountered in literature, namely the Engels' classification and the Simonds' classification.

2.1. *The Engels' Classification*

This classification proposed in [Engels *et al.* (2001)] is based on two orthogonal dimensions. The first dimension introduces the notions of *intra model* (or *horizontal)* and *inter model* (or *vertical)* inconsistencies.

- **Intra model (or horizontal) inconsistency:** Produced when the consistency is not validated between different models or submodels at the same level of abstraction. It means that they can overlap and sometimes include conflicting specifications. For example, in any UML model, the class diagrams and their associated sequence diagrams and state diagrams should be mutually consistent. So, for instance an operation may be (re)moved in a class diagram while an instance of this class (i.e.,

[a] The paper is an extended version of [Allaki *et al.* (2014)].

an object) in a sequence diagram still relies on this operation to handle a message it receives from another object [Straeten (2005)].

- **Inter model (or vertical) inconsistency:** Produced when the consistency is not validated between models at different levels of abstraction. Vertical consistency conflicts can result from refining a model or submodel, then it requires the refined model to be consistent with the one it refines [Straeten (2005)].

The second dimension deals with the syntactic and semantic inconsistencies.

- **Syntactic inconsistency:** Produced when there is no conformance of models to the abstract syntax of the modeling language. In the case of UML, this means that the user-defined UML models must conform to its abstract syntax. The abstract syntax of UML is a set of class diagrams described in the metamodels, in addition to some "well-formedness rules" expressed in OCL [Straeten (2005)].
- **Semantic inconsistency:** Produced when the behavior of models aren't semantically compatible. UML lacks formal semantics. Its semantics is described by OCL constraints and some informal statements in natural language (in English). UML is a general purpose modeling language, and hence even the informal specified semantics can change, depending for example on the modeling process used. UML specifications do not address semantic consistency issues at all, which make checking this type of inconsistences relatively difficult [Straeten (2005)].

The two dimensions above are orthogonal. Hence, they can be combined to form the Engels' inconsistency classification as shown in Table 1.

Table 1. The Engels' inconsistency classification: possible combinations.

Types of inconsistency	**Syntactic**	**Semantic**
Horizontal	X	X
Vertical	X	X

2.2. *The Simonds' Classification*

This classification proposed in [Simmonds (2003)] focuses on the conceptual aspects of the inconsistencies, and also suggests two orthogonal dimensions. The first dimension concerns the structural and behavioral aspects of the affected model.

- **Structural inconsistencies:** Arise when the structure of the system is inconsistent, and typically appear in class diagrams which describe the static structure of the system [Mens *et al.* (2005)].
- **Behavioral inconsistencies:** Arise when the specification of the system behavior is inconsistent, and typically appear in sequence and state diagrams that describe the dynamic behavior of the system [Mens *et al.* (2005)].

The second dimension is about the abstraction level (specification or instance) of the affected model.

- **Specification level:** Contains model elements belonging generally to class diagram. These elements such as classes and associations serve as specification for instances.
- **Instance level:** Contains model elements belonging generally to sequence or state diagrams. These elements like objects, links, transitions and events specify instances.

The two dimensions above are also orthogonal with possible overlapping for the abstraction level (specification and instance). Hence, they can be combined to form the Simonds' inconsistency classification as shown in Table 2.

Table 2. The Simonds' inconsistency classification: possible combinations.

Types of inconsistency	**Behavioral**	**Structural**
Specification	X	X
Specification - Instance	X	X
Instance	X	X

As a conclusion, we observe that no one of these two classifications gives attention to the nature of the inconsistency which can take different forms. Moreover, no one of these classifications gives information about how many diagrams are affected by the inconsistency. Also, we suggest combining these two classifications to have a comprehensive vision that encompasses the different cases seen from different viewpoints.

3. Our classification of model inconsistencies

Our proposed way to classify UML model inconsistencies is based on two kinds of classifications called respectively *terminological* and *typological*.

3.1. *The Terminological classification*

The terminological classification of inconsistencies focuses on the various meanings of the term "inconsistency", which can be *incompleteness*, *ambiguity*, *contradiction*, *incompatibility* and *anomaly* as shown in Fig. 1.

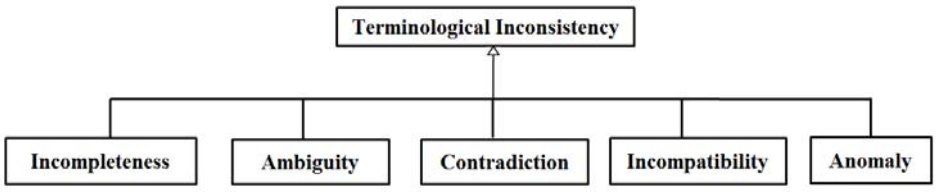

Fig. 1. Terminological classification.

- **Incompleteness:** Means that the presence of information in a model requires the presence of other information in other parts of the model. Otherwise, if we find

information that can be deduced from the current model and that are not present in it, then we talk about incompleteness [Lange *et al.* (2003)].

- **Ambiguity:** Means that interpretation of a model or a part of it in more than one way is possible [Fecher *et al.* (2005)].
- **Contradiction:** Means that there is a discrepancy between different parts of the model. Each part of it reflects something different.
- **Incompatibility:** Means that the model is not in agreement with the rules defined in the UML metaclasses.
- **Anomaly:** Means that the concerned part of the model can be improved in terms of the quality of the solution.

3.2. *The Typological classification*

The typological classification of inconsistencies is a unification of existing classifications into one hierarchy along several dimensions. These inconsistencies may concern either a single diagram (Mono-diagram) or several diagrams (Multi-diagrams).

3.2.1. *The Mono-Diagram inconsistencies*

The Mono-Diagram inconsistencies concern only one diagram with no relationship with others neither vertically nor horizontally. Indeed, they include the structural/behavioral and the specification/instance dimensions of the classification as presented in Section 2.2 in addition to the syntactic/semantic dimension of the classification as introduced in Section 2.1.

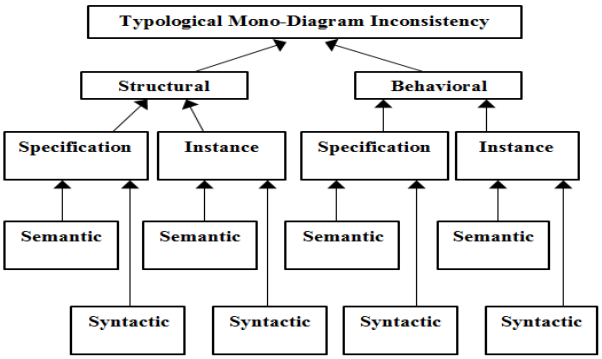

Fig. 2. Hierarchy of mono-diagram inconsistencies.

As shown in Fig. 2 mono-diagram inconsistencies can be structural or behavioral, which may be at specification or instance level. These ones can be either syntactic or semantic. For example, a “cyclic inheritance” which means a mutual relationship of inheritance between two classes A and B (A extends B, and B extends A) is a syntactic inconsistency since it is not allowed by the abstract syntax of UML. It is considered then as an *incompatibility* problem. However, when an attribute or a parameter type of an

operation in a class diagram refers to a non-existing class, then this example is considered as a semantic inconsistency. Further explanations are given in Section 4.

3.2.2. *The Multi-Diagram inconsistencies*

The multi-diagram inconsistencies concern the inconsistencies involving two diagrams or more either vertically or horizontally. They combine the dimensions of Mono-diagram inconsistencies (of Fig. 2) and the *intra model* (or *horizontal)* and *inter model* (or *vertical)* inconsistencies of the classification presented in Section 2.1. Thus, we can have a semantic horizontal inconsistency that concerns at the same time structural aspect at specification level, or a semantic vertical inconsistency that concerns behavioral aspect of the model at specification/instance level, and so on.
The hierarchy of multi-diagram inconsistencies is depicted separately in two parts (for clarity and readability reasons) as shown in Fig. 3 and Fig. 4.

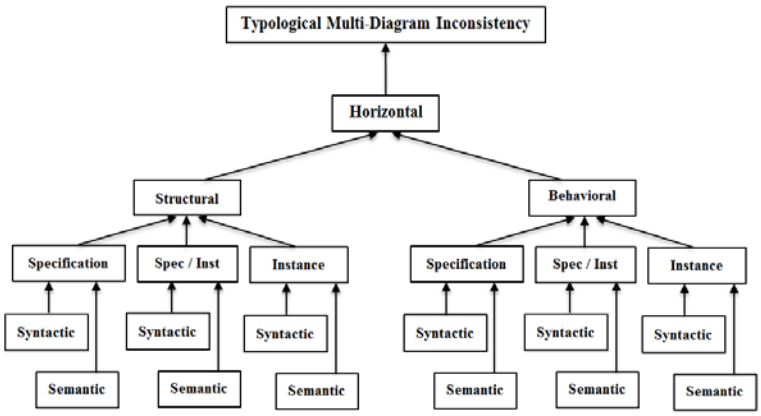

Fig. 3. Hierarchy of multi-diagram inconsistencies (part 1).

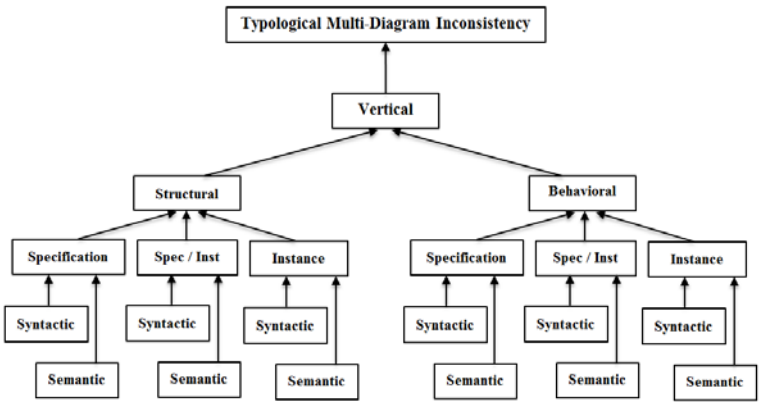

Fig. 4. Hierarchy of multi-diagram inconsistencies (part 2).

As shown in Fig. 3 the horizontal inconsistencies, either structural or behavioral, can take one of the abstraction levels (specification, instance or specification/instance) which can be either syntactic or semantic. The vertical inconsistencies in turn are presented in the same way in Fig. 4.

4. Inconsistency Examples

The various inconsistencies presented above are illustrated using some representative UML diagrams taken from an academic management system. The system consists of giving the possibility to manage the academic programs, courses, projects of a university, in addition to its students, professors and employees. An extract of the class diagram of this system is shown in Fig. 5.

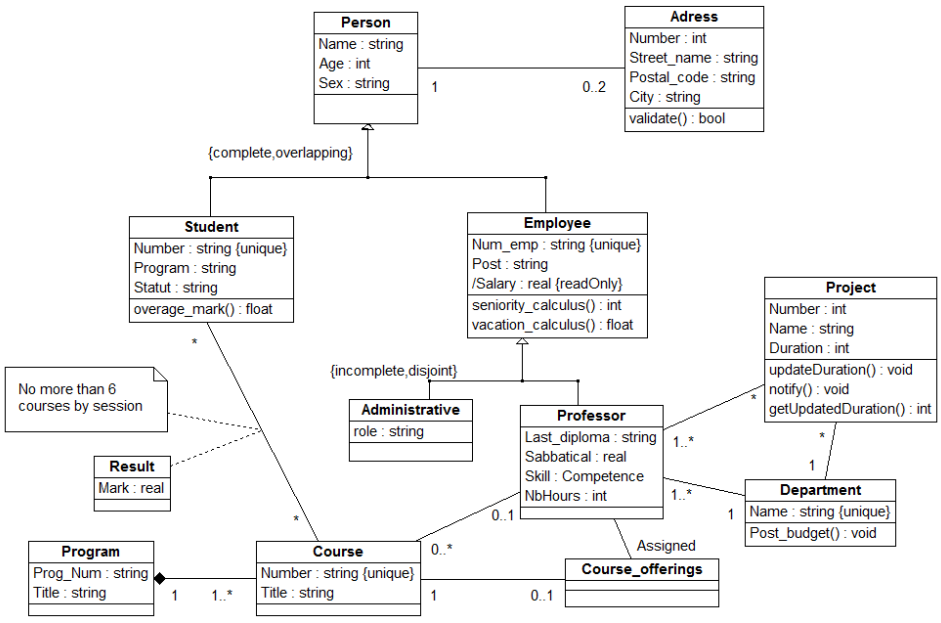

Fig. 5. A part of the Class diagram of an Academic Management System.

Example 1: Mono-diagram: Structural-Specification-Semantic Inconsistency

Fig. 6 shows the class “Professor” with some attributes and one operation.

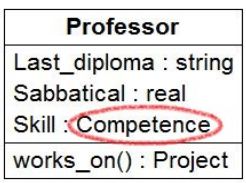

Fig. 6. Dangling type reference.

If the class diagram doesn't define the class "Competence", then the use of "Competence" as a type of the attribute "Skill" raises an *incompleteness* inconsistency. This inconsistency is considered as mono-diagram because it concerns precisely one diagram. The class diagram represents the specification level. The lack of an element of it influences the structural aspect of the model, which is correct syntactically but incorrect semantically.

The subsequent examples are related to multi-diagram inconsistencies. They focus on semantic inconsistencies. The syntactic ones are related to the violation of well-formedness rules of UML.

Example 2: Horizontal-Structural-Specification-Semantic Inconsistency

In this example, as mentioned in [Straeten (2005)], we make a distinction between sequence diagrams representing object interactions and the ones representing role interactions. The first ones belong to the instance level and the latter ones to the specification level. These sequence diagrams usually describe interactions between roles when their intention is to describe prototypical interactions which can be used as a patterns.

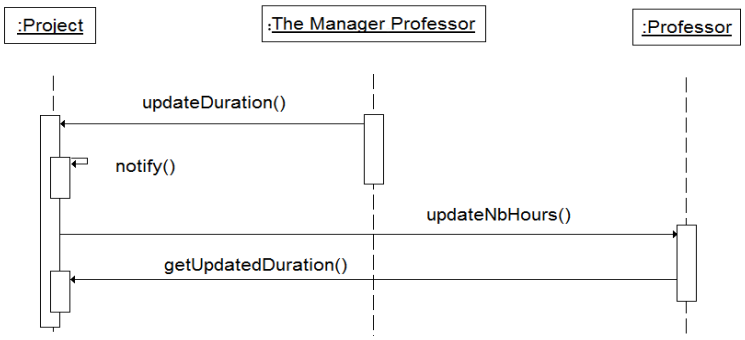

Fig. 7. Part of Sequence diagram at specification level.

Fig. 7 shows a part of a sequence diagram at the specification level. This figure shows a generic interaction between the roles "Project" and "Professor" showing that whenever the duration of any project is updated the hourly load (represented by NbHours) of each professor working on that project is automatically updated too. More precisely, when the project role receives message updateDuration(), then immediately notifies the concerned professors. This scenario can be used as a general pattern; it is also similar to the interaction ensured by the observer design pattern [Gamma *et al.* (1994)].

An inconsistency occurs if the message "updateNbHours" does not refer to an existing operation in the class "Professor" in the class diagram of Fig. 4. This *incompleteness* inconsistency is horizontal because it affects two diagrams at the same abstraction level. In addition, it concerns the specification level of the model and the missing operation affects the structure of the model. The syntax of this part of the model is correct due to its conformance to the adequate "well-formedness" rules. However, its semantics is incorrect as explained before.

Example 3: Horizontal-Structural-Instance-Semantic Inconsistency

Fig. 8 shows the state diagram of a particular project and Fig. 9 shows an activity diagram describing a simple process of project validation with an object flow showing the project object in various states: “Created”, “Validated”, and “In Progress”.

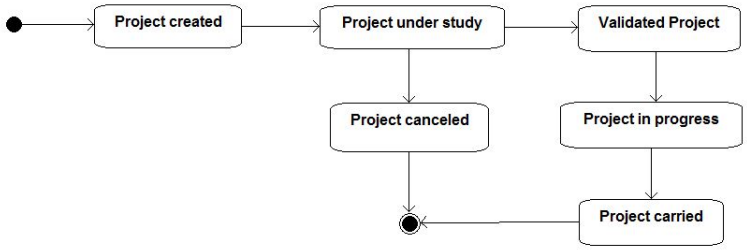

Fig. 8. State diagram of the class “Project”.

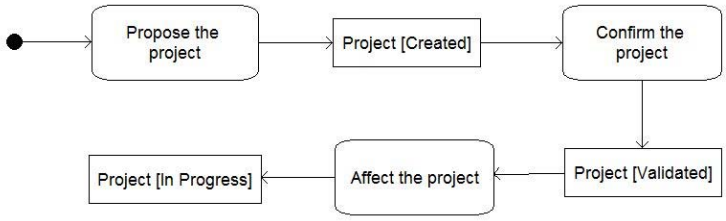

Fig. 9. Activity diagram for a simple process of project validation.

The activity diagram of Fig. 9 doesn’t consider the state “Project under study” introduced in the state diagram of Fig. 8. This can lead to a problem of accessibility to the state “Project canceled”.

This *incompleteness* inconsistency is horizontal because it affects two diagrams at the same abstraction level. The state and activity diagrams refer to objects of the instance level. The missing state in the activity diagram influences the structure of the model. Thus, the model is syntactically correct, but semantically incorrect.

Example 4: Horizontal-Behavioral-Instance-Semantic Inconsistency

Fig. 10 presents another example of an activity diagram related to the process of project validation showing the project object in the following consecutive states: “Under study”, “Created”, and “Validated”.

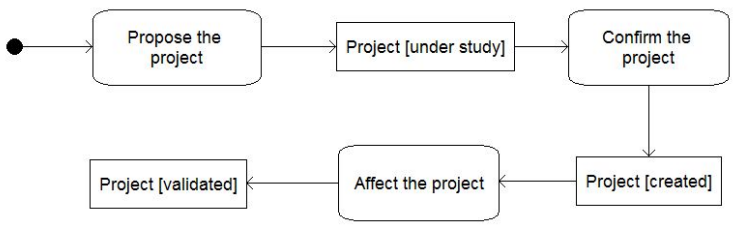

Fig. 10. Another activity diagram for a simple process of project validation.

This state succession doesn't match the state succession of the state diagram (Fig. 8). There is a contradiction between the two diagrams concerning the succession of the states "Project created" and "Project under study". In this case, the inconsistency is also horizontal, and it affects two diagrams at the instance level. The succession of states in the activity diagram concerns the behavior of the model which is syntactically correct but incorrect semantically.

Example 5: Horizontal-Behavioral-Specification /Instance-Semantic Inconsistency

Fig. 11 shows a simple sequence diagram at the instance level. It depicts an instance of class "Course" that sends the message "prog_list" to an instance of class "Program".

Fig. 11. A simple Sequence Diagram at instance level.

The message "prog_list" returns a set of programs while the relationship between classes "Course" and "Program" in the class diagram (of Fig. 5) shows that a "Course" belongs to exactly one "Program" (The multiplicity on the "Program" side is 1). Yet, it's preferable to avoid this *anomaly* and to ameliorate the solution by not using a collection in this case, and to be content with a single return type. This semantic inconsistency is horizontal since it involves two diagrams of the same abstraction level. The class diagram and the sequence diagram pair used represent the specification/instance of the model. The non-conformance regarding the multiplicity constraint is rather considered a behavioral inconsistency than a structural one.

Example 6: Vertical-Structural-Specification/Instance-Semantic Inconsistency

In this example, we consider that the part of the sequence diagram presented in Fig. 12 is a refinement of an existing sequence diagram at the instance level. The refinement is used to present more details on the interaction between the two objects used in this example. We assume also that the class diagram (Fig. 5) is on a higher level of abstraction than this sequence diagram.

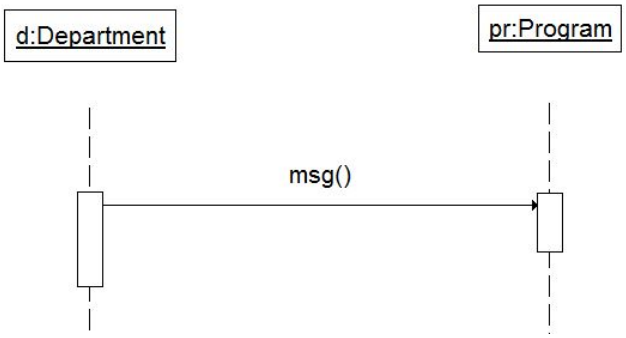

Fig. 12. Part of Sequence diagram at instance level.

Fig. 12 shows then an instance of "Department" sending a new introduced message "msg" to an instance of "Program" although in the class diagram of Fig. 5 there is no direct relationship between the two classes "Department" and "Program". This raises a *contradiction,* considered as a semantic inconsistency.

This inconsistency is vertical because it affects two different levels of abstraction: the class diagram and the refined sequence diagram. Like the previous example, the class and the sequence diagram pair used represent the specification/instance of the model. And the missing relationship between the classes "Department" and "Program" concerns the structure of the model.

5. Existing techniques for ensuring UML model consistency

In this section, we discuss about existing approaches and used techniques for checking UML model inconsistencies. These verification techniques can be classified into two categories: the formal approaches and the non-formal approaches. For each category, we provide hereafter a wide coverage of proposals found in literature. At the same time, we use our taxonomy of inconsistencies to specify in a table which ones are supported by these works.

5.1. *Formal techniques*

Initial researches of checking UML model inconsistencies suggest using an approach based on reasoning mechanisms of a formal language. The principle is to transform the semi-formal UML model and its consistency rules to any formal language. Then, inconsistencies are detected using inference mechanisms of the formal language. Those techniques are classified in this paper according to the underlying formal paradigm, as identified in [Habrias and Frappier (2006)]. We have three main paradigms: State transitions, Process Algebra and Logic.

5.1.1. *State transitions*

The principle of this paradigm is that the specification describes a relation of transition on a set of states. Many works dealing with the problem of checking inconsistencies use state transitions techniques. For instance, [Amalio *et al.* (2004)] and [Miloudi *et al.* (2011)] translate UML models to the Z language. [Kim and Carrington (2004)] in turn, choose to work with Object-Z. [Laleau and Polack (2008)] use the B language.
Likewise, [Fryz and Kotulski (2007)] provides a Graph based solution, while [Diethers *et al.* (2004)] suggest a Timed Automata (UPPAAL model checker) proposal. [Haesen and Snoeck (2004)] propose a MERODE methodology that provides a formal definition of UML diagrams. Other works choose to use the Petri Nets techniques like [Yao and Shatz (2006)]. In turn, [Shinkawa (2006)] opts for the Colored Petri Nets (CPN) and [Bernardi *et al.* (2002)] use the Generalized Stochastic Petri Nets (GSPNs).

5.1.2. *Process Algebra*

Process Algebra is a special type of algebra. Its operations are applied to elementary processes and events to describe how events may occur. The most popular works using this formal paradigm are [Engels *et al.* (2002a)], [Engels *et al.* (2002b)] and [Lam and Padget

(2005)]. The Engels' proposals are based on the Communication Sequential Process (CSP) techniques and Lam's work is based on the pi-calculus method. Also under the category of process algebra paradigm, we can consider the RT-LOTOS solution proposed by [Apvrille *et al.* (2004)].

5.1.3. *Logic*

In Logic-based paradigm, the behavior of functions is formalized using a set of equations (axioms) which state how functions are related. Among the most known works using this paradigm, there are [Zhao *et al.* (2006)] based on the SPIN model checker, [Malgouyres and Motet (2006)] based on Constraint Logic Programming (CLP), [Paige *et al.* (2007)] based on the Prototype Verification System (PVS), and [Straeten *et al.* (2007)] based on Description Logic (DL).

It should be noted in passing that some other works are based on combined techniques between different paradigms to check inconsistencies in UML models. For example, [Hee *et al.* (2004)] use Petri Nets and Z, [Rasch and Wehrheim (2003)] use Object-Z and CSP, and [Yeung (2004)] uses B and CSP.

By using formal techniques, we can take advantage from the underlying formal languages which give solid mathematical proof and have a mathematical foundation that offers numerous applications such as theorem provers, model checkers and coherence checkers. Also, the formal methods add more precision to UML models which are basically not as accurate.

5.2. *Non-formal techniques*

The second group of techniques deals with the problem of inconsistencies without using formal methods; we call them non-formal techniques. In most cases, the proposed solutions of this group are based on constraints. The principle is that the constraints are defined for UML metamodel. Then, the inconsistencies in UML models are detected in accordance with these constraints. Some of these solutions check the correctness of the model in just one aspect related to a single diagram. For instance, [Pakalnickiene and Nemuraite (2007)] and [Berkenkötter (2008)] use OCL rules. In turn, [Chen and Motet (2009)] propose controlling grammar in XML format "C-Control" for expressing correctness rules.

Other solutions define constraints among different aspects related to more than one diagram to check the model consistency. For example, [Kalibatiene *et al.* (2013)] provide a rule-based method for consistency checking in IS models. [Sapna and Mohanty (2007)] propose examples of OCL consistency rules and their translation to SQL. And [Egyed (2007)] in his work uses constraints defined with the language/checker available in IBM rational Rose.

It should also note here that additional non-formal techniques using other approaches exist. For instance, [Graaf and Deursen (2007)] propose manual inspection with a four steps transformation, [Elaasar *et al.* (2011)] propose a method to declaratively specify and automatically detect inconsistences using QVT, and [Amaya *et al.* (2006)] in their model

coding based approach, use the xLinkit tool which can also be used to verify models coded with XMI.

Most of the non-formal techniques mentioned above let us preserve all the information expressed in the UML model unlike the formal techniques which addresses only some aspects of UML diagrams, those that can be converted to the selected formal language. On the other hand, non-formal approaches add also more precision to the model expression and provides extensible proposals taking into account the usability and the maintainability of the solution.

Table 3 shows the different kinds of inconsistencies (see our taxonomy in Section 3) supported by both the formal and non-formal techniques referred to above.

Table 3. Inconsistencies supported by some solutions of ensuring UML models consistency.

Inconsistencies classification / Solutions					Formal techniques			Non-formal techniques		
					[Haesen and Snoeck (2004)]	[Engels *et al.* (2002a), (2002b)]	[Straeten *et al.* (2007)]	[Pakalnickiene and Nemuraite (2007)]	[Kalibatiene *et al.* (2013)]	[Graaf and Deursen (2007)]
Mono-Diagram		Structural	Specification	Syn				X		
				Sem						
			Instance	Syn				X		
				Sem						
		Behavioral	Specification	Syn				X		
				Sem						
			Instance	Syn				X		
				Sem						
Multi-Diagram	Horizontal	Structural	Specification	Syn	X	X	X		X	
				Sem	X	X	X		X	
			Spec/Inst	Syn	X	X	X		X	
				Sem	X	X	X		X	
			Instance	Syn	X	X	X		X	
				Sem	X	X	X		X	
		Behavioral	Specification	Syn	X	X	X		X	X
				Sem	X	X	X		X	X
			Spec/Inst	Syn	X	X	X		X	X
				Sem	X	X	X		X	X
			Instance	Syn	X	X	X		X	X
				Sem	X	X	X		X	X
	Vertical	Structural	Specification	Syn		X	X		X	
				Sem		X	X		X	
			Spec/Inst	Syn		X	X		X	
				Sem		X	X		X	
			Instance	Syn		X	X		X	
				Sem		X	X		X	
		Behavioral	Specification	Syn		X	X		X	
				Sem		X	X		X	
			Spec/Inst	Syn		X	X		X	
				Sem		X	X		X	
			Instance	Syn		X	X		X	
				Sem		X	X		X	

Table legend: **Syn:** Syntactic **Sem:** Semantic **Spec/Inst:** Specification/Instance

As shown in Table 3, some of these works focus on one aspect of the model (Mono-diagram) like [Pakalnickiene and Nemuraite (2007)], or many aspects (Multi-diagram). These multi-diagram methods provide support for either the Horizontal Behavioral inconsistencies such [Graaf and Deursen (2007)], or for all the Horizontal inconsistencies with the Structural and the Behavioral aspects like [Haesen and Snoeck (2004)], or both the Horizontal and Vertical inconsistencies with all their aspects as [Engels *et al.* (2002a)], [Engels *et al.* (2002b)], [Kalibatiene *et al.* (2013)] and [Straeten *et al.* (2007)].

6. Conclusion

The paper first presented an overview of existing classifications of UML model inconsistencies, and then proposed a new way to classify UML model inconsistencies based on two kinds of classifications called respectively terminological and typological. The terminological classification of inconsistencies focuses on the various meanings of the term "inconsistency" which can be incompleteness, ambiguity, contradiction, incompatibility and anomaly. The typological classification of inconsistencies, however, is a unification of existing classifications in one hierarchy along several dimensions. These inconsistencies may concern either a single diagram (Mono diagram) or several diagrams (Multi diagram). Both mono and multi diagram inconsistencies can be structural or behavioral, may be at specification or instance level and can be either syntactic or semantic. Unlike mono-diagram inconsistencies, the multi-diagram inconsistencies can be furthermore horizontal (concerning different diagrams at a same level of abstraction) or vertical (concerning different diagrams at different abstraction levels). Some representative examples were given to illustrate the various combinations of inconsistencies.

Beside the classification of inconsistencies, we gave in this paper an overview of the existing approaches ensuring UML models consistency, and classified them into two categories. The first one, called formal approaches, concerns the transformation of semi-formal UML models to formal languages using formal paradigms like state transitions, process algebra or logic. The second one, called non-formal approaches, uses rather other techniques such for example the notion of constraints to check the model consistency. In addition to this classification of inconsistency verification approaches, we also specified through a table which of the inconsistencies, among those figuring in our taxonomy, are supported by the existing works for ensuring UML models consistency.

As future work, we intend to develop a CASE tool for automating the identification and the verification of all the inconsistencies presented in this paper by defining some constraints at the metamodel level. The CASE tool will also include modules for fixing the inconsistencies found.

References

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.

Amálio, N.; Stepney, S.; Polack, F. (2004): Formal proof from UML models, in: Formal Methods and Software Engineering. In Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings, pp. 418–433.

Amaya, P.; Gonzalez, C.; Murillo, J.M. (2006): Towards a subject-oriented model-driven framework. In Electronic Notes in Theoretical Computer Science, Volume 163, Number 1, September 2006, pp. 31-44.

Apvrille, L.; Courtiat, J.-P.; Lohr, C.; de Saqui-Sannes, P. (2004): TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. In IEEE Transactions on Software Engineering (TSE), Volume 30, Number 7, July 2004, pp. 473-487.

Berkenkötter, K. (2008): Reliable UML Models and Profiles. In Electronic Notes in Theoretical Computer Science, Volume 217, July 2008, pp. 203-220.

Bernardi, S.; Donatelli, S.; Merseguer, J. (2002): From UML Sequence Diagrams and Statecharts to Analyzable Petri Net models. In WOSP 2002: Third International Workshop on Software and Performance, July 24-26, 2002, Rome, Italy, pp. 35-45.

Chen, Z.; Motet, G. (2009): A Language-Theoretic View on Guidelines and Consistency Rules of UML. In Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), Twente, Netherlands, Lecture Notes in Computer Sciences n° 5562, Springer (June 2009), pp. 66 - 81.

Diethers, K.; Huhn, M. (2004): Vooduu: Verification of Object-Oriented Designs Using UPPAAL. In Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, pp. 139–143.

Dubauskaite, R.; Vasilecas, O. (2013): Method on specifying consistency rules among different aspect models, expressed in UML. In Electronics & Electrical Engineering; 2013, Vol. 19 Issue 3, pp. 77-81.

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.

Elaasar, M.; Briand, L.; Labiche, Y. (2011): Domain-Specific Model Verification with QVT. In Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings, pp. 282-298.

Engels, G.; Küster, J.M.; Heckel, R.; Groenewegen, L. (2001): A methodology for specifying and analyzing consistency of object oriented behavioral models. In Proceedings of Eighth European Software Engineering Conference Held Jointly with Ninth ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE2001), September 2001, ACM Press, Vienna, Austria,12,58, pp.186–195.

Engels, G.; Küster, J.M.; Heckel, R.; Groenewegen, L. (2002a): Towards consistency preserving model evolution. In IWPSE02 International Workshop on Principles of Software Evolution, Orlando, FL, USA, May 19-20, 2002, pp. 129–132.

Engels, G.; Heckel, R.; Küster, J.M.; Groenewegen, L. (2002b): Consistency-preserving model evolution through transformations. In UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings. pp. 212–226.

Fecher, H.; Schönborn, J.; Kyas, M.; de Roever, W.P. (2005): 29 new unclarities in the semantics of UML 2.0 state machines. In Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings, pp. 52-65.

Fryz, L.; Kotulski, L. (2007): Assurance of system consistency during independent creation of UML diagrams. In International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX 2007), June 14-16, 2007, Szklarska Poreba, Poland, pp. 51–58.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 395p.

Graaf, B.; Deursen, A.V. (2007): Model-Driven Consistency Checking of Behavioral Specifications. In Model-based Methodologies for Pervasive and Embedded Software, 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MOMPES 2007, Braga, Portugal, March 31, 2007, Proceedings, pp. 115-126.

Habrias, H.; Frappier, M. (2006). *Software Specification Methods*. (Eds) London, UK: ISTE, 418p.

Haesen, R.; Snoeck, M. (2004): Implementing Consistency Management Techniques for Conceptual Modeling. In Third International Workshop on Consistency Problems in UML based Software Development III Understanding and Usage of Dependency Relationships, October 2004, Lisbon, Portugal, pp. 99-113.

Hee, K.M.V.; Sidorova, N.; Somers, L.J.; Voorhoeve, M. (2004): Consistency in model integration. In Business Process Management: Second International Conference, BPM 2004, Potsdam, Germany, June 17-18, 2004. Proceedings, pp. 1–16.

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.

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.

Kim, S-K.; Carrington, D.A. (2004): A Formal Object - Oriented Approach to defining Consistency Constraints for UML Models. In 15th Australian Software Engineering Conference (ASWEC 2004), 13-16 April 2004, Melbourne, Australia, pp. 87- 94.

Laleau, R.; Polack, F. (2008): Using formal metamodels to check consistency of functional views in information systems specification. In Information & Software Technology, Volume 50, Numbers 7-8, June 2008, pp. 797–814.

Lange, C.; Chaudron, M.R.V.; Muskens, J.; Somers, L.J.; Dortmans, H.M. (2003): An empirical investigation in quantifying inconsistency and incompleteness of UML designs. In 2nd workshop on consistency problems in UML-based software development, October 20, 2003, in San Francisco, USA as part of the "International Conference on Unified Modeling Language 2003", pp. 26-34.

Lam, V.S.W.; Padget, J.A. (2005): Consistency checking of sequence diagrams and statechart diagrams using the pi-calculus. In Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings, pp. 347–365.

Lucas, F.J.; Molina, F.; Toval, A. (2009): A systematic review of UML model consistency management. In Information and Software Technology, December 2009, Volume 51, Issue 12, pp. 1631–1645.

Malgouyres, H.; Motet, G. (2006): A UML model consistency verification approach based on meta-modeling formalization. In Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006, pp. 1804–1809.

MDA Guide Version 1.0.1, <http://www.omg.org/mda>, 2003. (Last accessed January 2014).

Mens, T.; Straeten, R.V-D.; Simmonds, J. (2005): A framework for managing consistency of evolving UML models. In Software Evolution with UML and XML, chapter 1. Idea Group Inc., March 2005, pp. 1-30.

Meta Object Facility, <http://www.omg.org/mda>, 2011. (Last accessed January 2014).

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.

Noraini, I.; Rosziati, I.; Zainuri, S.M.; Dzahar, M.; Tutut, H. (2011): Consistency rules between UML use case and activity diagrams using logical approach. In International Journal of Software Engineering & Its Applications, Jul 2011, vol. 5 Issue 3, pp. 119-134.

Paige, R.F.; Brooke, P.J.; Ostroff, J.S. (2007): Metamodel-based model conformance and multiview consistency checking. In ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 16, Number 3, July 2007.

Pakalnickiene, E.; Nemuraite, L. (2007): Checking of conceptual models with integrity constraints. In Information technology and control, Volume 36, Number 3, 2007, pp. 285–294.

Rasch, H.; Wehrheim, H. (2003): Checking consistency in UML diagrams: classes and state machines. In Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003, Proceedings, pp. 229-243.

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.

Schmidt, D. (2006): Guest editor's introduction: model-driven engineering. In IEEE Computer Society, February 2006, Volume 39, No. 2, pp. 25-31.

Shinkawa, Y. (2006): Inter-model consistency in UML based on CPN formalism. In 13th Asia-Pacific Software Engineering Conference (APSEC 2006), 6-8 December 2006, Bangalore, India, pp. 411-418.

Simmonds, J. (2003). *Consistency maintenance of UML models with description logics.* Unpublished master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France. 2003.

Straeten, R.V.D. (2005). *Inconsistency management in Model-Driven Engineering; an approach using description logics.* Ph.D. thesis, System and Software engineering lab, Department of Computer Science, faculty of science, Vrije Universiteit Brussel, Belgium. September 2005.

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.

Unified Modeling Language: Superstructure. Version 2.4.1, Retrieved from: <http://www.omg.org/spec/UML/2.4.1/>, 2011. (Last accessed January 2014).

XML Metadata Interchange, <http://www.omg.org/spec/XMI/2.4.1/>, 2013. (Last accessed January 2014).

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.

Yeung, W.L. (2004): Checking consistency between UML class and state models based on CSP and B. In The Journal of Universal Computer Science, Volume 10, Number 11, 2004, pp. 1540–1559.

Zhao, X.; Long, Q.; Qiu, Z. (2006): Model checking dynamic UML consistency. In Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings, pp. 440–459.