Building Consistent UML Models for Better Model Driven Engineering

Driss Allaki¹, Mohamed Dahchour², Abdeslam En-nouaary³
¹,²,³ National Institute of Posts and Telecommunications
Rabat, Morocco
d.allaki@inpt.ac.ma
dahchour@inpt.ac.ma
abdeslam@inpt.ac.ma

ABSTRACT: The OMG (Object Management Group) has designed its own vision of the emerging software paradigm MDE (Model Driven Engineering) under the name of MDA (Model Driven Architecture). For this aim, OMG proposes a set of standardized rules and practices to promote good modeling and perfectly exploit models. To fully benefit from this approach, the UML models, to be used within MDA, should be consistent. In this paper, we point out that inconsistencies could be a serious problem that negatively affects the sustainability, the productivity and the consideration of execution platforms aimed by MDA. As a solution, we propose a constraint-based method that defines consistency rules expressed using EVL (Epsilon Validation Language) to automatically detect and fix the potential inconsistencies that could compromise the application of MDA through UML diagrams.

1. Introduction

Software engineering has known over the last few decades the development of several leaps namely the third generation languages, the structured programming, the component-based software engineering and so on. The common characteristic between every new leap of software development is the improvement in the abstraction level with the aim to approach the human thinking development methodologies. As a result, this software evolution tends to ensure a certain platform independence, which makes the software more portable.

In particular, MDE (Model Driven Engineering), which is seen as an emerging software paradigm, absorbs the main improvements of the previous software leaps and adds a model-based development methodology that brings a higher abstraction level by relying on models as primary artifacts to build complex software systems [1].

In order to make MDE the next established way to develop software in the near future, OMG (Object Management Group) added more relevance to this concept and proposed as early as 2000 its own vision of MDE under the name of Model Driven Architecture [3]. MDA as cited in [2] is “an OMG initiative that proposes to define a set of non-proprietary standards such as UML [4], MOF [5] and XMI [6] that will specify interoperable technologies with which to realize model-driven development with automated transformations.”
Indeed, MDA recommends the development of the following three complementary models that refer to all the development stages of software from requirements’ elicitation to implementation and testing.

- **CIM (Computation Independent Model):** is a complex entity in which no computer consideration is taken into account. It is consisting of a glossary, definitions of business processes, requirements and use cases as well as a systemic view of the application. The purpose of these models is to define which services are offered by the application and what are the other entities with which it interacts.

- **PIM (Platform Independent Model):** is the abstract analysis and design model. MDA considers that these models must be independent of any platform implementation. They are typically expressed as a large collection of interdependent and partially overlapping UML diagrams. Their role is to be sustainable and to link between requirements model and the code of the application. They also need to be productive; i.e., they must be sufficiently precise and contain sufficient information for a possible automatic code generation.

- **PSM (Platform Specific Model):** is the concrete design model. They primarily serve to facilitate the code generation from an analysis and design model and they are platform-specific.

To fully benefit from MDA, the different models that contain adequate information for automated transformations (i.e., PIMs) should be consistent, or free of inconsistencies. The term inconsistency, in this context, is a state in which one or more diagrams of UML model portray conflicting [7]. These conflicts between UML diagrams could be syntactic or semantic contradictions, ambiguities, incompleteness or anomalies [8] and may be caused by the multi-view nature of software systems or by the incremental and iterative nature of the used software development process [9].

A lot of inconsistencies in one model could cause many errors and anomalies in software design. As they can also, in a larger context, invalidate the MDA models and complicate the model transformation process. As a result, the expected benefits of MDA could be directly affected by the different fallouts of the problem. For instance, the model’s sustainability expected by MDA will have no value if the software system information we want to keep perennial is represented in an incomplete, ambiguous or contradictory model since this latter could easily lose its expressiveness and then its ability to ensure the information sustainability. Besides, the productivity gains and the consideration of execution platforms also aimed by MDA could in turn be affected by inconsistencies. In fact, MDA integrates models into the production process of the application within the model transformations. However, operating these models cannot be done in a proper way if they do not contain correct and sufficient information for a possible automatic code generation. Furthermore, many basic operations applicable to models such as merging, matching, slicing, and splitting are increasingly automated to generate productivity gains throughout the different stages of the application life cycle. However, these operations can cause some problems like duplication of some elements or non/multiple matches for others.

These statements are confirmed by the experimentation presented in [10] which show that the absolute number of inconsistencies in UML designs is quite large; and that in industrial practice, designs are moved into implementation stage while there are still significant numbers of inconsistencies in the model. This sort of practices can increase the software development complexity or can downright lead to the project termination [11].

To address this issue, engineers, researchers and practitioners have proposed a large set of methods, frameworks, tools, theories and good practices for checking inconsistencies. However, some limitations related to the maturity and to the practicability of these proposals create some hurdles that prevent totally solving the problem once and for all.

This paper proposes a method that deals with the inconsistencies in UML models at the meta-level using Epsilon Validation Language (EVL) [28]. The method consists of defining a set of EVL constraints to detect and fix inconsistencies when modeling a software system in an MDA context.

The rest of the paper is organized as follows. Section II introduces the whole idea and the key features used in our constraint-based method for checking UML model inconsistencies. Section III gives an overview of the most known verification techniques found in literature and provides a discussion about the relevance of our consistency checking proposal. Section IV concludes the paper and presents future work.

### 2. Our constraint-based method to deal with inconsistencies in UML models

In this section, we present the main contribution of the paper. We first introduce the constraint-based approach to deal with model inconsistencies. Then, we present our method using EVL constraints illustrated with motivating inconsistency examples.

#### 2.1 The constraint-based approach

Using constraints to check UML model inconsistencies has widely been used in many consistency checking methods (for instance [23], [24], [25] and [26]). The main idea behind this technique is to define formal constraints at the meta-level of UML in order to be applied to the UML artifacts. This technique follows a *consistency by monitoring* strategy [12] by describing different conditions that UML models have to satisfy to be considered
consistent. Then, checking inconsistencies will be based on detecting violations of UML models’ consistency according to these defined constraints since any UML model inherits all the specifications, including constraints, from its metamodel. Also, the consistency constraints defined at the UML metamodels have the advantage of being independent from any specific implementation platform which allows them to be generically applied to all UML models.

On another side, these methods are based on defining a set of constraints, considered as consistency rules, to deal with all the types of inconsistencies from both the syntactic and the semantic aspects, either being mono-diagram (arising within a single diagram) or multi-diagrams (arising between a couple of diagrams). In both cases, it is required to match between different metaclasses of the different diagrams by establishing the right links when defining a consistency constraint.

2.2 Ensuring UML model’s consistency through EVL constraints

As depicted in Figure 1, the consistency in UML diagrams is ensured according to the instances of formal constraints expressed in EVL. For example, if a UML model contains an inconsistency occurring in one diagram, or among two or more of its diagrams, an instance of the formal EVL constraint is activated when validating the model to inform the modeler via a message that an inconsistency was detected. In fact, the instance of the EVL constraint takes action in accordance to the consistency rule defined at the meta-level. The consistency rule acts on the concerned metaelements and the adequate meta-associations linking them in the UML metamodel.

To well implement the method, we exploit the strengths of the Epsilon technologies. In fact, Epsilon is a set of languages and tools for code generation, model-to-model transformation, model validation, and other model’s operations that work with EMF and other types of models in an MDE context [15]. EVL is a task-specific language of the general model management language Epsilon [13]. EVL is dedicated to validate models. In their simplest form, constraints expressed in EVL are quite similar to OCL constraints. However, unlike OCL, EVL supports many other features such as:

- Dependencies between constraints (e.g. if constraint A fails, don’t evaluate constraint B).
- Support of user interaction (quick fixes for failed constraints).
- Customized error messages to be displayed to the user.
- Distinction between errors and warnings during validation.

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

2.3 Building EVL constraints

To build a constraint that deals with an inconsistency, there are a number of steps to be followed as illustrated by the **Dangling Operation** inconsistency example [14] depicted in Figure 2 and Figure 3.

The **Dangling Operation** inconsistency arises if a message refers to an operation, in the class diagram, that does not belong to the class attached to the receiving event of the message in the sequence diagram lifeline.

As illustrated in Figure 3, the part of the sequence diagram represents an instance of class “A” sending a new introduced message “m2()” to an instance of class “B”. However, the message “m2()” refers to an operation in class diagram illustrated by Figure 2 that does not belong to the class “B” attached to the receiving event of the message in sequence diagram.

![Figure 1. The principle of checking inconsistencies using constraints in the meta-level](image1)

![Figure 2. A part of a class diagram](image2)

![Figure 3. A part of a sequence diagram](image3)
In fact, when adapting models, 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.

In order to formalize the textual description of the Dangling Operation example, we have to first define the UML elements involved in this consistency conflict, which are in our case: Message, Operation, Class and Lifeline.

The next step is to determine, according to the UML specification, how these elements' metaclasses are linked. In our example, the links between the main involved metaclasses are illustrated in Figure 4.

![Figure 4. Metaclasses involved in the Dangling Operation inconsistency](image)

To define the EVL constraint, it is required to identify the adequate metaclass, from those involved in the inconsistency, to be the context of the constraint. A context specifies the kind of instances on which the contained constraint will be evaluated [15]. The metaclass “Message” is the identified context of our example.

```plaintext
context Message
```

Each EVL context contains an invariant that defines a name. Its declaration can differ according to how we precise the critical level of the inconsistency. This feature provided by the EVL lets us distinguish between errors and warnings when defining the invariant. Errors, declared in EVL invariant as “constraint”, indicate critical deficiencies that contradict and invalidate the modeled artifacts. However, warnings, declared in EVL invariant as “critique”, indicate non-critical issues that should even be addressed by the modeler [15].

Considering our inconsistency example as an error leads us to declare it as follows:

```plaintext
constraint DanglingOperation
```

In addition to the name, each EVL invariant defines a body (check). In our example, we consider for clarity reasons the simplest instance of the Dangling Operation in which we have one operation in the class. Then, the body of our constraint compares 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. The inconsistency occurs when the two signatures are different. The defined body (check) is then presented as follows:

```plaintext
check :
self.receiveEvent.covered.represents.type.ownedOperation -> includes (self.signature)
```

The failure reasons can be defined in the message part. In our example, the message to display if the inconsistency arises is:

```plaintext
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.”
```

EVL allows repairing detected inconsistencies using Epsilon Object Language [16]. Concerning our example, the actions contained in the fix part create a new operation in the corresponding class of the message.

```plaintext
fix { title : “add an operation to the class”
do { var op = new Operation;
op.name = self.name;
sel.receiveEvent.covered.represents.type.ownedOperation.add(op);
}
```

These were the steps to follow in order to define an EVL constraint for the Dangling Operation inconsistency example. The same steps are followed to define the constraint for any other type of inconsistency as shown in Figure 5.

To summarize, Figure 6 shows the interactive process for detecting and resolving design defects in a UML model using our method. Once a modeler finishes creating his UML model, he can validate it via the used tool. The detection of the possible inconsistencies is based on the EVL constraint body (check). If no inconsistencies are detected, the process finishes successfully. Else, the descriptive message related to the detected inconsistency is displayed in the tool output. This message proposes some resolution actions that are based on those already defined in the EVL constraint. The user then chooses the appropriate fixing action that will be applied to the model. The same process is repeated until the model is free of inconsistencies.
Figure 5. Building the EVL constraints process

Figure 6. The interactive process for detecting and resolving inconsistencies in a UML model

Figure 7. A part of the Class diagram of an Academic Management System
2.4 Illustrative examples
In addition to the Dangling Operation inconsistency detailed above, more examples on how to deal with inconsistencies using our method are presented in this section. In this regard, we consider the case study from [8] that represents an extract of a UML model (PIM) expressing the functioning of an Academic Management System. We assume that this system is built using a model-driven development allowing the management of 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 Figure 7.

Based on the exhaustive list of inconsistency examples presented in the systematic literature review carried out in [29], we limited ourselves to the three following inconsistencies for illustrative purposes.

a) The Navigation Incompatibility:
The sequence diagram presented in Figure 8 contains a message of which calling direction does not match the navigation constraint on the corresponding association. More explicitly, the message \( msg() \) is sent from a sender object (instance of the Address class) to a receiver object (instance of the Student class) in opposition to the navigation direction of the association between the two corresponding classes (Address and Person “super class of Student”) in the class diagram.

We can effectively detect such inconsistencies by using our EVL constraint-based method.

The involved metaclasses of this inconsistency are shown in Figure 9.

Figure 8. A part of a sequence diagram that triggers the Navigation Incompatibility

![Diagram of Navigation Incompatibility](image_url)

Figure 9. Metaclasses involved in the Navigation Incompatibility
The EVL constraint that checks this inconsistency is presented as follows:

```java
context Message {
    constraint NavigationIncompatibility{
        check:
        self.receiveEvent.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 metaclass 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.

**b) The Connector Type Incompatibility:**

As illustrated in Figure 10, an instance of the `Department` class sends a new introduced message `msg()` to an instance of the class `Program` although there is no direct relationship between the two classes in the class diagram (Messages only between related classes).

The involved inconsistency elements from the UML metamodel are shown in Figure 11.

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

```java
context Connector {
    constraint ConnectorTypeIncompatibility{
        check:
        self.type.member End->includesAll(self.end.definingEnd
        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"
    }
}
```

![Figure 10. A part of a sequence diagram that triggers the Connector Type Incompatibility](image)

![Figure 11. Metaclasses involved in the Connector Type Incompatibility](image)
In this example, we choose the metaclass Connector as a context for the constraint detecting the Connector Type Incompatibility. We make sure if the types of the connectable elements that the ends of the connector are attached include 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.

c) The Multiplicity Incompatibility:

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

```plaintext
context Connector {
    constraint MultiplicityIncompatibility{
        check : self.end.lower < self.end.definingEnd.lower 
or self.end.upper > self.end.definingEnd.upper
        message : "The model contains a connector end" 
+ self.end.name + "of which the multiplicity is 
more general than the multiplicity of the 
association typing the owning connector"
    }
}
```

The Multiplicity Incompatibility is detected using a constraint that uses the metaclass Connector as a context and that compares the meta-attributes lower and upper of the metaclasses Property and ConnectorEnd. If the constraint is not respected, an explanatory message is displayed.

Therefore, adding these EVL constraints to the UML metamodel prevent the arising of any occurrence of these inconsistencies by detecting them at early stages of the design process.

3. Related work

Over the last decade, checking UML model inconsistencies has been a priority research investigation in both model-based and model-driven development. As a consequence, several contributions have been devised to deal with this issue. As explained in [16], these contributions may be classified into many categories according to the typological nature of the solution. We mainly mention among them:

• Tools, technologies and applications that can be used to identify and resolve UML model inconsistencies.
• Processes that consist of a set of linked procedures which take one or more resources to convert inputs into outputs until gaining certain outcome on ensuring models' consistency.

• Theories or set of statements that describe logical facts on checking inconsistencies, especially those repeated several times in different environments.

• Frameworks and models used to describe UML model validation entities or process in a hypothetical manner.

• Methods, procedures and techniques used to check the consistency with accuracy and efficiency.

• Algorithms which help solving consistency problems in UML.

Note that other types of contributions that provide mixed approaches could be found in literature (e.g. Algorithm & Process, Method & Tool, etc.).

These contributions could also be categorized according to another classification based on the consistency strategy of the solution. We generally adopt one of the two following strategies: the consistency by construction and the consistency by monitoring.

Techniques based on construction strategy generate one artifact from another. They are founded on detecting inconsistencies after transforming semi-formal UML models to a formal language, using inference mechanisms of the targeted language. They generally provide solid mathematical proof, foundation and tools and make UML models more precise by avoiding ambiguities when handling inconsistencies in these models. Hereafter, we present some of the works based on transformation to formal languages.

Hanzala and Porres [17] present an automatic tool chain implementing UML to Web Ontology Language (OWL 2) translations. They describe how to translate UML class, object and statechart diagrams in OWL 2 and present a reasoning of these UML models using logic reasoners for OWL2. The proposed solution is fully automatic and does not require any expertise about the targeted language and its reasoners from the designer.

Cabot et al. [18] present a method to check UML class diagrams extended with OCL constraints. The method translates the UML/OCL model into a Constraint Satisfaction Problem (CSP) to deal with complex invariants which may include numerical constraints.

Singh et al. [19] propose a practical approach that transform the use case diagram, class diagram and sequence diagram to Z schema for capturing both the syntax and semantics for safety critical systems.

Straeten et al. [20] show how the reasoning capabilities of Description Logics (DL), a decidable fragment of first-order logic, can be used in a natural way to detect behavior inconsistencies of a refined model, and also detect the behavior preservation violations during model refactoring.

Yao and Shatz [21] ensure consistency of UML dynamic diagrams using Petri Net techniques. They focus on translating sequence and statechart diagrams with composite states to an Extended Colored Petri Net used to describe state transitions of individual objects and interactions among them.

Zhao et al. [22] carry out an investigation of a SPIN model checker based approach to check the consistency between the sequence and statechart diagrams. They propose a variant of automata, called Split Automata, to efficiently translate the statechart diagrams to SPIN in order to deal with the hierarchy structure of statechart diagrams.

Overall, construction-based techniques provide solid mathematical foundation and add more precision to UML models by taking advantage from the underlying formal languages. In contrast, they generally do not support all the properties of the UML models since they are limited only to those properties that can be expressed and analyzed by the target formal language of the translation. Furthermore, most of the construction-based techniques and languages are not commonly known among the industrial development communities. They are difficult to use due to the lack of tool support and documentation in addition to the strong mathematical background required for applying them.

On the other hand, techniques based on monitoring strategy are primarily based on formal constraints defined at the UML metamodel. Techniques using this approach are extensible, since they give the possibility to include new consistency rules for new arising inconsistencies. They also make the model more expressive and preserve all the information expressed through different UML diagrams.

Przigoda et al. [23] help designers to be aided during the debugging of their model by suggesting an automatic method which comprehensibly analyzes constraints causing contradictions and determines reasons explaining the contradiction in an inconsistent UML/OCL model.

Kalibatiene et al. [24] provide a rule-based method for consistency checking in IS models, which is implemented to check consistency in UML diagrams. The consistency rules expressed in this work associate meta-elements from different aspects of models despite the fact that they are directly associated in a metamodel.

Reder and Egyed [25] propose in their work an automated and tool supported approach that increase the performance of re-validating design rules. This re-validation takes place as a result of the large model changes caused by different model's operations.
Hamman et al. [26] try in their work to handle behavioral model's aspects using OCL. They show that protocol state machines can take advantage of OCL in form of OCL state invariants and OCL guards and post-conditions for state transitions.

Sapna and Mohanty [27] deal with structural inconsistencies between the use cases, activity, collaboration, statechart and class diagrams by using OCL rules converted to SQL triggers applicable across tables that store the UML diagrams.

Unlike construction-based techniques, most of the monitoring-based proposals have the advantage of preserving all the information expressed in the UML models being studied. They also add more precision to the model by making it more expressive through the constraints defined at the metamodel. Moreover, they take into account the possibility to include new consistency checks for new arising inconsistencies. Nevertheless, constraint-based techniques suffer from a lack of formality. Also, if the constraints used do not cover all the potential inconsistencies some of these might left undetectable in the models. Finally, existing monitoring-based techniques generally deal with static aspects of the UML models and are limited to checking inconsistencies in a single diagram, which can compromise their efficiency.

Giving this overview of the existing inconsistency checking techniques, we recall that our method exploits the strengths of the Epsilon technologies to bring the following characteristics in the modeling area:

a) Our constraint-based method is conceived in accordance to the given EVL functionalities to automatically detect inconsistencies in UML models, provide customized error messages and propose actions and quick fixes to repair these errors. Gathering these three functionalities in one solution is the main distinctive feature of our method.

b) The extensible nature of the method gives the possibility to add new constraints to deal with any new arising inconsistency and then to be complete in terms of coverage of both potential inconsistencies and the UML diagrams commonly used.

c) EVL, the technology used to implement the defined constraints on the UML metamodel, is in accordance to the other Epsilon languages and tools for code generation, model-to-model transformation and further MDE operations on models.

d) The monitoring strategy of our method allows enabling constraints once the modeler explicitly asks the validation of his model and not during modeling stage. Then, some “false positives” – fake inconsistencies that are inevitably produced when the model is under construction – could be avoided.

e) The solution is easily automated using Eclipse Modeling Platform independent models (PIMs) has many negative impacts on the MDA approach, as well as on other broader Model Driven Engineering activities. In fact, it is of utmost importance to ensure the consistency of these models (generally expressed with UML) to take full advantage from the MDA approach in terms of expertise sustainability, productivity gains and consideration of execution platforms.

The main contribution of this paper is the proposition of the concepts of a monitoring-based proposal that help building consistent UML models by detecting and fixing inconsistencies according to defined EVL constraints at the meta-level. This method aims to make the modeler’s experience simpler and more interesting by automatically detecting inconsistencies, providing helpful error messages and proposing actions and quick fixes to repair the detected UML model’s inconsistencies at early stages of an MDE process.

Motivated by the need of including more suitable repairing actions, we intend to deeply work in the future on handling inconsistencies. We will focus on generating a set of small, complete and correct repair actions by eliminating false and non-minimal repairs. Also, we plan to investigate mathematical techniques based on decision theory to help the designer select the more appropriate alternative from the generated repair actions. A comprehensive case study that contains patterns involving a set of tricky examples of inconsistencies and usage rules that cover a larger
number of expressive UML diagrams will be used to initially evaluate and discuss the new version of the entire solution.

References


