Detecting Semantic Equivalence in UML Class Diagrams

Valéria Oliveira Costa¹ ²
1 Instituto Federal de Educação, Ciência e Tecnologia do Piauí, IFPI
Teresina, Brasil
valeria@ifpi.edu.br,

Rodrigo S. Monteiro², Leonardo G. P. Murta²
1 Instituto de Computação
Universidade Federal Fluminense, UFF
Niterói, Brasil
{salvador, leomurta}@ic.uff.br

Abstract— When developing a system in Model-driven Engineering (MDE), developers change the same diagram in parallel. These changes generate different versions that may conflict. Conflicts can be syntactic, related to the structure of the diagram, or semantic, related to the meaning of the diagram. The detection of semantic conflicts in diagrams should take into consideration both the syntax and semantics. This is necessary because languages like UML provide different representations that produce the same meaning. Therefore, syntactically different diagrams may be semantically equivalent. Thus, conflict detection methods must understand the semantics of diagrams to reduce the occurrence of false positive and false negative conflicts. This paper presents a semantic conflict detection method for UML class diagrams. It verifies if two versions of a class diagram are semantically equivalent. It performs the syntactic comparison and checks if the two diagrams contain the same elements with the same semantics. If they do, the diagrams are considered semantically equivalent.

Keywords-component; model version control; semantic equivalence; semantic conflict detection; UML class diagram.

I. INTRODUCTION

The UML aims at assisting developers by providing tools to analyze, design, and implement systems [1]. One of the key diagrams in UML is the class diagram. A class diagram provides a static view of a system and consists of classes and relationships (association, aggregation, generalization, realization, etc.). According to [2], class diagrams are widely used for modeling systems and have several different structures with the same meaning. For example, we can represent a relationship amongst classes by means of an association or an attribute, setting its type as the opposing class, or even mixing both representations [3]. Therefore, the choice of which representation to use is the responsibility of the modeler [2].

When the development team uses Model-driven Engineering (MDE), the development of a system is done through the creation, modification, and evolution of models [4]. In this context, a model can contain or be restricted to a class diagram. During the system development, different versions of diagrams are generated in parallel by developers. This occurs for different reasons that range from time-to-market to isolation of evolutive and corrective maintenance. These versions need to be controlled to ensure consistency. In this scenario arises the need of Model-driven Version Control System (VCS). A Model-driven VCS stores the versions of models and provide support for model comparison, detection and resolution of conflicts, and merging. Among the existings Model-driven VCS we can mention Odyssey-SCM [5], AMOR [6], and Mirador [7].

Model-driven VCS must check for the occurrence of conflicts amongst versions. Conflict is a set of contradictory changes where at least one operation performed by a developer does not comply with at least one operation performed by another developer [8]. A good conflict detection method should minimize the occurrence of false positive and false negative conflicts. The former are non-conflicting changes marked by the detection method as conflicts [9]. The latter are conflicting changes not marked by the detection method as conflicts [9]. False positivies conflicts reduce the productivity of developers since the time taken to analyze them could be used for other tasks. False negatives conflicts produce even more harmful consequences, as they may be ignored by the development team, leading to inappropriate merges.

When the conflict detection method considers only the syntax of diagrams, it is restricted to the detection of syntactic conflicts. These conflicts are detected by structural comparison between the versions of diagrams [10][11]. As the method does not recognize the semantics of diagram, equivalent representations can be diagnosed as conflicting (false positives conflicts). Understanding the semantics of the diagram allows the identification of different representations that have the same meaning. Thus, the method becomes able to ascertain whether syntactically different versions are semantically equivalent. This helps in reducing false positives conflicts.

Moreover, understanding the semantics of diagram allows the detection of semantic conflicts. A semantic conflict occurs when changes performed in parallel not only interfere with the modified element, but also in others. For example, semantic conflicts may occur when a developer modifies an element that depends on another element modified by other developers [10]. As semantic conflicts are more difficult to detect, they can generate false negative conflicts. Therefore, understanding the
semantics of the diagrams can also help reducing false negative conflicts.

This paper presents a semantic equivalence detection method for class diagrams, called Rainbow, to solve the aforementioned problems. Our method investigates the semantic equivalence of class diagrams in order to reduce false positive and false negative conflicts. It receives an original version and two revisions of it. These three versions are exported as Prolog facts and enriched with semantic rules. Afterwards, the differences between the original version and its revisions are computed. These differences consist of the items added or deleted in a revision from original version. The differences are checked for semantic equivalence and semantic conflicts are detected. In a previous work [12], we present our semantic conflicts detection method for UML use case diagram. In this paper, we extended the method for UML class diagram.

The remaining of the paper is organized as follows: section II presents important concepts about Model-driven VCS; Section III explains how the Rainbow works; Section IV shows an example of Rainbow in action; Section V discusses the technologies used to implement a tool that automates the method; Section VI presents some related works; and Section VII presents the conclusion and future work.

II. BACKGROUND

Rainbow assumes a development environment that uses MDE and an optimistic Model-based VCS. As we want to add no extra burden to the development environment, we also assume that the VCS is loosely coupled to it. This way we focus on a three-way state-based version control [13].

In three-way state-based version control, the merge algorithm receives three versions: the version that the developer wants to put in the repository (developer), the latest version stored in the repository (current), and the common ancestor version from which both derived (base). In this approach, in general, developers change the model on their workspace, independently and in parallel, until they decide to socialize their contributions [13]. When the developer takes this decision, he or she sends their developer version to the VCS. Then, the VCS checks for conflicts with the current version and generates a merged version.

The complete process of the three-way merge has four phases [14]: comparison, conflict detection, conflict resolution, and merge. The comparison phase contrasts the developer and current versions in order to calculate their differences. The information contained in base version is taken into consideration to help inferring the operations performed in each version. This phase is critical to the final result of the merge process because the other phases are based on information collected by it. In state-based merge, the comparison algorithm relies only on the input versions (base, current, and developer) to compute the differences between versions. Therefore, the method requires matching techniques to check if an element corresponds to another in different versions.

Next, the conflict detection phase verifies whether the differences contain conflicts. There are various types of conflicts such as syntactic and semantic conflicts. This work focuses on the detection of semantic conflicts. During the analysis of differences, if developer and current versions are equal or semantically equivalent then there are no conflicts. In this case, the merge phase can be performed. Otherwise, the existing conflicts must be resolved (conflict resolution phase) before continuing the merge process. After resolving the conflicts, the merge phase can be performed. At this phase, the base and developer versions are combined into a single version (merged version).

III. RAINBOW

Rainbow considers a subset of relationships and elements that exist in the specification of the UML class diagram (CD) [1]. This subset contains the following elements: class, attribute, operation, multiplicity, role, association, aggregation, composition, interface, realization, generalization, enumeration, and abstract class.

Rainbow is composed of three phases as shown in Fig. 1 [12]. These phases are: translation, semantic enrichment, and conflict detection. In the translation phase, our method receives the three versions of CD to be analyzed (base, current, and developer versions) and transforms them into a set of Prolog facts. Each Prolog fact represents an element or relationship in a CD version. Tab. I shows some examples of Prolog facts generated in this phase.

Table I. Mapping From Class Diagram to-Prolog Facts

<table>
<thead>
<tr>
<th>Diagram class item</th>
<th>Prolog fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>class(className).</td>
</tr>
<tr>
<td>Abstract class</td>
<td>isAbstractClass(className).</td>
</tr>
<tr>
<td>Generalization</td>
<td>inheritance(subclassName, superclassName).</td>
</tr>
<tr>
<td>Attribute of class</td>
<td>attribute(className, attributeName).</td>
</tr>
</tbody>
</table>

During the semantic enrichment phase, Rainbow analyzes each Prolog fact generated during translation phase to infer indirect relationships. This inference generates new Prolog facts that are added to those created in the previous phase. To accomplish this, the class diagram specification has been
analyzed and we generated a set of rules that describes the semantics of the class diagram according to the UML metamodel. Tab. II shows some specific semantic rules for the UML class diagram used by Rainbow.

Next, the conflict detection phase starts. Rainbow compares the enriched set of Prolog facts to identify if they are semantically equivalent or if one version semantically contains the other. If one of these two alternatives is true, it means that there are no semantic conflicts. If the first alternative is true, it means that any of the versions can be chosen. If the second alternative is true, then the versions should be analyzed for occurrences of semantic conflicts.

The semantic conflict detection consists of executing two diff operations. The first is performed between base and developer versions while the second is performed between base and current versions. Each diff operation computes the additions and deletions that transformed the base version in the version in comparison. The additions form the add set and the deletions form the del set.

### TABLE II. SEMANTIC RULES FOR UML CLASS DIAGRAM

<table>
<thead>
<tr>
<th>#</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In a generalization relationship, for each attribute in the superclass, an equivalent attribute is inferred for the subclass unless the attribute in question has a private visibility.</td>
</tr>
<tr>
<td>2</td>
<td>In a generalization relationship, for each operation in the superclass, an equivalent operation is inferred for the subclass unless the operation in question has a private visibility.</td>
</tr>
<tr>
<td>3</td>
<td>In a generalization relationship, for each association in the superclass, an equivalent association is inferred for the subclass.</td>
</tr>
<tr>
<td>4</td>
<td>In a realization relationship, for each attribute of the interface, an equivalent attribute is inferred for the class.</td>
</tr>
<tr>
<td>5</td>
<td>In a realization relationship, for each operation of the interface, an equivalent operation is inferred for the class.</td>
</tr>
<tr>
<td>6</td>
<td>If a class is abstract and all its methods are abstract then, if there is an interface that has the same name and the same elements, they are semantically equivalent.</td>
</tr>
</tbody>
</table>

Next, the add and del sets of each diff operation are checked for conflicts. A conflict occurs if a model element appears simultaneously in the set of additions of the first diff operation and in the set of deletions of the second diff operation, or vice versa. For a better understanding of the functioning of Rainbow, section IV presents an example of our method in action.

### IV. RAINBOW IN ACTION

Consider the development of a system for bank control, and the modeling of their CD. Fig. 2 shows three simplified versions for such CD. Fig. 2.a shows the base version, Fig. 2.b presents the current version, and Fig 2.c shows the developer version. In the proposed method, all three illustrated versions have their files submitted to the translation phase as explained in Section III. After the execution of this phase, three Prolog sets are generated, representing all relevant information about the input versions. During this phase, only the syntax of CD is considered. Due to space constraints, this paper does not show all the generated facts during the translation phase.

Next, the semantic enrichment phase is performed. In this phase the Prolog facts are enriched with semantic rules specific for UML CD. These rules infer indirect relationships through direct relationships amongst CD elements, conciliating the semantics of different representations.

In order to illustrate the semantic enrichment performed by the semantic rules, consider the figures that compose Fig. 2, more specifically Fig. 2.a and 2.b. Analyzing the differences between these two figures, we can observe that the developer made the following changes: he or she created an abstract class, named Person, and added a generalization between Natural Person and Person classes. He or she chose to put the attribute name and operation setName in the superclass. Another change made in this version was the addition of the association, named has, between Person and Account classes.

Furthermore, he or she added setSocialSecurityNumber operation in Natural Person class.

![Figure 2. Model Versions for Bank Control System.](image-url)
With respect to Fig. 2.b, the generalization between Natural Person and Person classes determines that Person is the superclass of Natural Person. Due to that, Natural Person inherits all attributes and operations that exist in Person whose visibility is not private. The same is also true for the associations.

These details justify the creation of new facts that represent these implicit behaviors (semantics) of the UML CD. Thus, in addition to the syntactic facts created in the translation phase, new semantic facts are created for both current and developer version from semantic rules shown in Tab. II.

Regarding the current version, by applying rule 1, the name attribute is inferred to Natural Person class and now this class has two attributes: name and socialSecurityNumber. Furthermore, by applying rule 2, the operation setName is inferred to Natural Person class and now this class has two operations: setName and setSocialSecurityNumber. Through the application of rule 3, associations are also inherited from Person. Then Natural Person is associated with class Account.

Consider now the figures 2.a and 2.c. Fig. 2.c was created from Fig. 2.a. Checking for changes made by the developer, it can be noticed that he or she created the Person interface. In Object Oriented paradigm, interfaces are public and cannot be instantiated. These structures specify contracts. Thus, the class that implements an interface contains all elements contained in that interface. In UML, the interface implementation is represented by a realization relationship.

Checking Fig 2.c, it is possible to note that the developer created a realization relationship between the Natural Person class and the Person interface. This relationship indicates that the Natural Person undertakes to implement the Person interface. When there is a relationship of realization between a class and an interface, the class in question contains not only the attributes specified in the class itself, but also the attributes and operations contained in the implemented interface.

These changes lead to the inference of new Prolog facts from UML semantic rules shown in Tab. II. Once again, it is worth mentioning that these new facts are added to the syntactic facts created in translation phase. In this case, these new facts were appended to the developer version by applying the semantic rules 4 and 5 of Tab. II. Through the application of rule 4, the Natural Person class has facts that correspond to name and socialSecurityNumber attributes. The name attribute comes from the Person interface. By rule 5, the setName operation also is inferred to Natural Person class.

After the end of the semantic enrichment phase, Rainbow starts the conflict detection phase. Analyzing this scenario, we can observe that two developers changed in parallel the same class diagram. Then, we need to check if these changes generated equivalent or conflicting versions.

| TABLE III. ADD SETS FOR CURRENT AND DEVELOPER VERSIONS |
|----------------|----------------|
| Current         | Developer      |
| class(person).  |                |
| visibilityclass(person,public). |             |
| isabstractclass(person). |              |
| inheritance(natural_person,person). |            |
| attribute(person,name). |              |
| visibilityattribute(person,name,person). |           |
| lowervalueattribute(person,name,1). |             |
| uppervalueattribute(person,name,1). |             |
| operationperson(setname). |             |
| typereurn(person,setname). |             |
| visiblityoperation(person,setname,person). |           |
| typeoperationinterface(person,setname). |           |
| operationparameter(person,setname,name,string). |         |
| operation(natural_person,setsocialsecuritynumber ). |          |
| typereurn(natural_person,setsocialsecuritynumber,unset). |         |
| operationparameter(natural_person,setsocialsecuritynumber,setsocialsecuritynumber,integer). |       |
| association(has,person,account). |             |
| aggregationassociation_end_a(has,person,person). |          |
| aggregationassociation_end_a(has,person,none). |            |
| lowervalueassociation_end_a(has,person,1). |            |
| uppervalueassociation_end_a(has,person,1). |            |
| roleassociation_end_b(has,account,account). |          |
| aggregationassociation_end_b(has,account,none). |          |
| lowervalueassociation_end_b(has,account,1). |            |
| uppervalueassociation_end_b(has,account,-1). |            |
| isnavigableassociation_end_b(has,account). |            |
In this phase, Rainbow executes the diff operations in the two pairs: base and current versions, and base and developer versions. For each pair, the add and del sets are computed. First, we consider the first pair (base-current diffs). As previously mentioned, the current version has only additions. Therefore, the del set is empty. Tab. III shows the result of this diff operation and contains all differences found in this scenario. The Add\textsubscript{Current} column presents the result of the diff operation that calculates the additions made in the current version in relation to base version.

Considering the second pair formed by base and developer versions, it is possible to notice that, as in the first pair, the developer only added elements to the base version. Therefore, the del set is also empty. The add set is shown in Tab. III in the Add\textsubscript{Developer} column.

After the creation of the add sets, the differences reported must be analyzed. In summary, one can say that the current version:

a) Created Person class, which is an abstract and public class. This class has an attribute called name and an abstract operation called setName. The subset of facts featured in Tab III, shows the specification of these concepts.

b) Added a generalization between Person and Natural Person classes. In this relationship Person is superclass of Natural Person.

c) Added an operation called setSocialSecurityNumber and an association between Person and Account class.

On the other hand, one can say that the developer version:

d) Added an interface called Person. This interface has an attribute called name and a setName operation. The setName operation is public and abstract.

e) Added a realization between the Person interface and Natural Person class.

Now, consider the Prolog facts represented in Table III. In Add\textsubscript{Current} column, the highlighted facts correspond to the a) and b) differences mentioned above. Likewise, in Add\textsubscript{Developer} column, the highlighted Prolog facts correspond to the d) and e) differences found in the developer version.

By rule 6 of Tab. II, it is concluded that Person interface and Person abstract class are equivalent since all methods of this class are abstract and public. Moreover, the name attribute has the same characteristics in both the interface and abstract class. The same happens with the setName operation. Thus, the Prolog facts that match the Person abstract class and all its elements, as well as those related to the Person interface, should be removed from both sets of difference. This removal operation makes the Add\textsubscript{Developer} set empty.

From this information, one can conclude that the current version includes all semantic which exists in the developer version. Therefore, Rainbow chooses the current version because this version is the most complete.

This type of semantic equivalence is not detected when only the syntax of the CD is analyzed. When only the syntax of the CD is considered, the method can identify these differences as conflicts and generate false positive conflicts. Also, not taking into account the semantic equivalence can lead to erroneous merged version.

As shown in the example, Rainbow contributes to identify syntactically different CD versions presenting the same semantics. This helps to reduce the amount of false positive conflicts and increases the efficiency of the method of conflict detection. This feature further reduces the rework generated for the team.

Moreover, understanding the semantics allows for detecting semantic conflicts. This type of conflict cannot be detected when only the syntax is considered. Thus, this detection can help reducing false negative conflicts. This also prevents erroneous merged version.

V. PROTOTYPE IMPLEMENTATION

We are implementing a tool to automate the Rainbow method. It currently supports the translation phase and we are implementing the semantic enrichment phase. When completed, Rainbow implementation will be expanded to other UML diagrams and, in the future, we will check for conflicts amongst different diagrams such as class diagrams and use cases.

The example of Section IV had its CD versions designed using Papyrus\textsuperscript{4}. For each CD version, Papyrus generates a file containing its XML Metadata Interchange (XMI) file. This file is used as input for Rainbow and translated into Prolog facts. We adopted the OMG Model to Text (M2T) standard to transform XMI files into a set of Prolog facts. The translation is made by means of Acceleo\textsuperscript{2}. Finally, the semantic enrichment phase uses TuProlog\textsuperscript{3} library, integrated to Java, to infer new Prolog facts.

VI. RELATED WORK

Odyssey-VCS [15] is a Model-driven VCS that allows the use of fine granularity for UML 2 models. The system uses Three-Way merge and state-based versioning. The conflict detection uses existence analysis of elements and processing of attributes and relationships. This VCS takes into account only the syntax of UML models. As it does not take into account the semantics of the models, it cannot detect semantic equivalence and semantic conflicts.

In [16] is proposed a domain-specific language that defines and manages syntactic and semantic conflicts. In this approach, the differences between the compared versions are represented in models of difference. The difference metamodel is generated from the base metamodel. For each metaclass of the base metamodel are generated three new classes: AddedMC, DeleteMC and ChangeMC. These artifacts aim at representing the added, deleted and modified classes of a version. It is applied the same reasoning for attributes, associations, methods and parameters. Moreover, the approach uses criteria resolution written in OCL (Object Constraint Language) and rules. This approach detects and resolves previously known semantic conflicts. However, we are unable to identify how semantic equivalence is treated.

Smover [17], presents a method of semantic conflicts detection. Smover is based on semantic views and inspection strategies of elements. The latter can be configured by the user. A semantic view maps a version from a source metamodel to target metamodel. This mapping is based on

\textsuperscript{4}http://www.eclipse.org/papyrus/
\textsuperscript{2}http://www.eclipse.org/acceleo/
\textsuperscript{3}http://tuprolog.alice.unibo.it/
relevant aspects of the source. The result of the transformation is a model in conformance with the target metamodel that contains the aspects of interest. The conflicts found in the source metamodel are syntactic conflicts, and those found in the mapped metamodel are semantic conflicts.

CDDiff [18] is an semantic diff operator created for UML CD. The operator receives two versions and executes its semantic comparison. The result of this operation is a set of diff witnesses. Each diff witness is an object model that is possible in the first version and is not possible in the second version. This operator uses a Two-Way merge, therefore does not detect semantic conflicts.

VII. CONCLUSION

This paper presents a semantic equivalence detection method for class diagrams called Rainbow. This method uses the three-way merge. Thus, it receives three versions and verifies if they are equivalent, if one version contains the other version, and if there are conflicts to be resolved.

Each submitted version is transformed into Prolog facts. These Prolog facts are semantically enriched by means of UML specific rules. Next, conflict detection phase checks for occurrences of semantic conflicts.

We also present an example that shows that syntactically different versions can be semantically equivalent. Due to the difficulty in identifying this type of equivalence, the method helps on reducing the amount of false positives conflicts. Thus, it increases the efficacy of the conflict detection method as a whole.

Moreover, understanding the semantics allows for detecting semantic conflicts. This type of conflict can not be detected when only the syntax is considered. Thus, this detection can help to reduce false negatives conflicts. This prevents that false negatives conflicts are not resolved which can lead to an erroneous merged version.

As future work, we intend to support additional UML diagrams (beyond the CD and use cases diagrams). Furthermore, we intend to expand the method to work in detecting conflicts between different diagrams. Finally, we are planning to run some experimental studies with the proposed method. We also intend to perform experiments on real case studies. Another future work is to allow the visualization of conflicts, changes, and merge by the developer.

ACKNOWLEDGMENT

We would like to thank Rhaylson Silva do Nascimento for his valuable contribution during the implementation phase of this work.

VIII. REFERENCES