# Semantic Conflicts Detection in Model-driven Engineering

Valéria Oliveira Costa[1,2], João M. B. Oliveira Junior[1], Leonardo Gresta Paulino Murta[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Piauí, IFPI
Teresina, Brasil
valeria@ifpi.edu.br
joaomanoel@aluno.ifpi.edu.br

[2] Instituto de Computação
Universidade Federal Fluminense, UFF
Niterói, Brasil
leomurta@ic.uff.br

***Abstract*— An important challenge of Model Driven Version Control System (VCS) is to use conflict detection methods that are appropriate for models. Methods that analyze only the syntax of models can detect conflicts that do not exist in reality (false positives) and can fail to detect conflicts that do exist (false negatives). This paper presents a method to reduce the occurrence of both false positive and false negative conflicts. For this, the presented method provides an analyzer of the semantic equivalence between models. Our method verifies if the model versions are semantically equivalent, if one version semantically contains the other version, and if there are conflicts between versions.**

***Keywords- model-based version control; semantic conflict detection***

## I. INTRODUCTION

With the advent of the Model-driven Engineering (MDE), which aims to facilitate the development of systems through the creation, manipulation, and maintenance of models, it became possible to direct the focus of the developers to design applications at higher abstraction levels [1]. Thus, a system can be constructed through the refinement of models that begins at the highest level of abstraction and goes toward the lower levels through the use of transformations [1].

In the context of MDE, during the development or evolution of a system, multiple versions of a model can be generated. Similarly to source code, this brings the necessity of model-driven Version Control Systems (VCS). A VCS helps the development team to manage the evolution of a software product through consistent maintenance of its many variants and revisions [2]. Therefore, a model-driven VCS manages model version. To do so, it compares models, detects and resolves conflicts, and makes the consistent merge of models. Among the existing Model driven VCS we can mention Odyssey-SCM [3], Smover [4], and Mirador [5].

One of the key concepts in the area of version control is the conflict. Conflict is a set of contradictory changes where at least one operation performed by the first developer does not agree with at least one operation performed by a subsequent developer [6]. A conflict is not desirable because it generates an additional effort for the developer [7] , which means rework. One might think that the time and effort needed to resolve a conflict could be used to continue the development or evolution of the system.

In this scenario, it may be noted that one important challenge of the model-driven VCS is the use of a conflict detection method that is appropriate and efficient to models. According to [8], in order to obtain success in a model merge process, it is necessary to understand not only the logical structure of the model, but also its semantics. According to [9], to resolve the conflicts, it is needed to identify the reasons of the conflict. This is especially difficult when only syntactic detection support is used.

In a modeling process, there may be situations where the same intention can be modeled in different ways. Thus, two developers working in parallel may use different strategies to model the same situation. A purely syntactic analyzer identifies this difference as conflicting. After a manual analysis, it can be verified that the conflict does not proceed, since the two representations are semantically equivalent. This type of conflict is false positive and reduces the efficacy of the conflict detection method, since the method should not report erroneous information to the developer.

One solution to reduce occurrences of false positive conflicts is to understand the semantics of the models. This understanding allows the identification of related syntactic conflict that is actually a semantic equivalence. Furthermore, a semantic analyzer also allows it to detect semantic conflicts. These conflicts occur when modifications in a given model element interfere in another model element even without explicit syntactic relationships among them. Semantic conflicts are more difficult to detect and, because of it, they generate false negatives conflicts. These, in turn, are conflicts that exist in reality, but unfortunately the conflict detection method cannot diagnose them.

This way, a good method of conflict detection should be able to identify semantic equivalences and not report them as

conflicts. This contributes to the reduction of false positive conflicts. Also, it must be able to detect semantic conflicts, thereby decreasing the number of undetected conflicts or false negatives conflicts.

To help solve the problems related above, this paper presents a semantic conflict detection method for models. The method focuses on the investigation of semantic equivalence of models in order to reduce false positive conflicts. It also uses the semantic understanding of the models to increase the coverage of the conflict detection method. The increased coverage decreases the likelihood of a semantic conflict undetected by the method (false positives).

The rest of the paper is organized as follows: section II presents important concepts about model-based version control; Section III explains the proposed method; Section IV shows an example of our method in action; Section V discusses the technologies used in the implementation of the prototype; Section VI presents some related works; and Section VII presents the conclusion and future work.

## II. BACKGROUND

Consider a scenario where the development team uses an optimistic VCS. With this type of VCS, each developer can work in an individual copy separately [2], until they decide to socialize their copy with other developers. Initially, the original version of the project, called *base version*, is stored in the repository. Then, all team members download the *base version* and start working on it separately. During socialization, the developer´s version, called *developer version*, should be analyzed and compared with the *base version*, and with the latest version already committed into the repository, called *current version*. The Fig. 1 shows this scenario. The process that considers the information contained in the ancestral version for calculating differences between two versions is used in three-way merge [2].

Another concept used in this work is state-based merge [2]. In this type of merge, only the information contained in the *base version* and its revisions are considered [2]. In this type of merge, there are no records of the operations performed that facilitate the understanding of the transformation from one version to another. On the other hand, this type of merge is more realistic, as it imposes no restriction over the development environment. The complete state-based merge process is composed of four phases [9]: comparison, conflict detection, conflict resolution, and the merge itself.

## III. SEMANTIC CONFLICTS DETECTION OF MODELS

This work is based on the before-mentioned state-based merge process. However, as it focuses only on the identification of semantic conflicts, it is restricted to the first two phases: comparison and conflict detection.

In the comparison phase, our method receives the three versions to be analyzed (*base*, *current* and *developer* versions). The versions are automatically transformed into a set of Prolog facts. Then, each Prolog version is analyzed in order to infer indirect relationships. This step is done by a Prolog set of rules that describes the semantic relationships of models according to

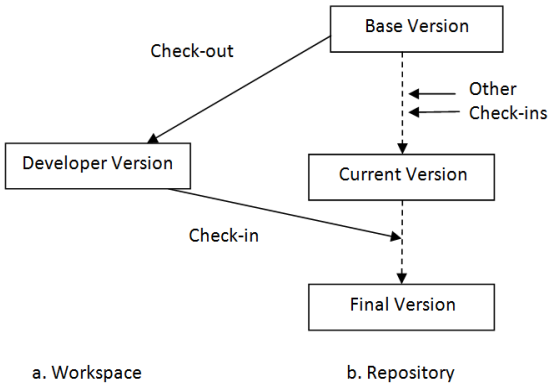



Figure 1. Development scenario

a specific metamodel. The versions are then compared to verify if they are semantically equivalent or if one version semantically contains the other. If the versions are not equivalent, and if the *current* version does not contain the *developer* version (or vice versa), then the conflict detection is initiated. In this phase, four sets are created considering the *base*, *current*, and *developer* versions: all model elements added to *base* version to compose *current* and *developer* versions and all model elements deleted from *base* version to compose *current* and *developer* versions. The special intersection among the sets of added and deleted model elements of different versions (*current* and *developer*) indicates conflicts.

The Fig. 2 provides an overview of the proposed method. In the first activity, called Translation, the *developer* version represents a version just produced by a member of the development team as a revision of *base* version. On the other hand, *current* version is the tip of the repository, created due to a previous commit performed by other team members. This way, *current* and *developer* versions were created in parallel, and both are revisions of *base* version. The goal of this activity is to transform these three versions into Prolog facts. Each fact refers to an existing element or relationship in the model. In the case of a relationship, there is involvement of a pair of elements in general. This way, the Translate activity result is a set of Prolog facts that represents all relevant syntactic information contained in the analyzed model.

The second activity, called Semantic Enrichment, is responsible for inferring new facts. To do so, this activity combines the previously generated facts with Metamodel-specific Rules. The Metamodel-specific Rules represent the semantics of relationships in a given metamodel. This way, they need to be set once and can be used for every model compliant to the metamodel. For instance, in the context of Use Case diagrams (UCD) of the UML metamodel, Tab. I shows the semantic rules used by our method. Such rules help on extracting semantics from the syntactic set of model elements represented as Prolog facts.

TABLE I. USE CASE DIAGRAM SEMANTIC RULES

| Semantic rules | |
|---|---|
| # | Rule |
| 1 | $\forall a, b \in Actor, \forall c \in UseCase,$ $inheritance(a,b) \wedge association(b,c) \Rightarrow association(a,c)$ |
| 2 | $\forall a \in Actor, \forall b, c \in Use\ Case,$ $inheritance(b,c) \wedge association(a,c) \Rightarrow association(a,b)$ |
| 3 | $\forall a \in Actor, \forall b, c \in Use\ Case,$ $include(b,c) \wedge association(a,b) \Rightarrow association(a,c)$ |
| 4 | $\forall a \in Actor, \forall b, c \in Use\ Case,$ $extend(b,c) \wedge association(a,c) \Rightarrow association(a,b)$ |

Next, the third activity, called Conflict Detection is performed. In this activity, the enriched Prolog facts of *current* and *developer* versions are analyzed and compared to *base* version. The analysis is done based on the three-way diff concept, forming two pairs: the first involving the diff between *base* and *current* versions and the second involving the diff between *base* and *developer* versions. For each pair, two sets of differences are computed: one that holds all added items (Add) and another that holds all deleted items (Del). These sets are computed according to (1) and (2):

$$Add_v = v \setminus base \quad (1)$$

$$Del_v = base \setminus v \quad (2)$$

Where $v \in \{current, developer\}$. After the computation of the additions and deletions sets, a conflict is detected if an element of the model appears simultaneously in the set of additions of the first pair and in the set of deletions of the second pair or vice versa. It is important to emphasize that to generate the set of semantic conflicts, the Metamodel-specific Rules were previously used during the Semantic Enrichment activity. Equation (3) denotes how the set of conflicts is formed:

$$Conflict_{v1,v2} = (Add_{v1} \cap^* Del_{v2}) \cup (Add_{v2} \cap^* Del_{v1}) \quad (3)$$

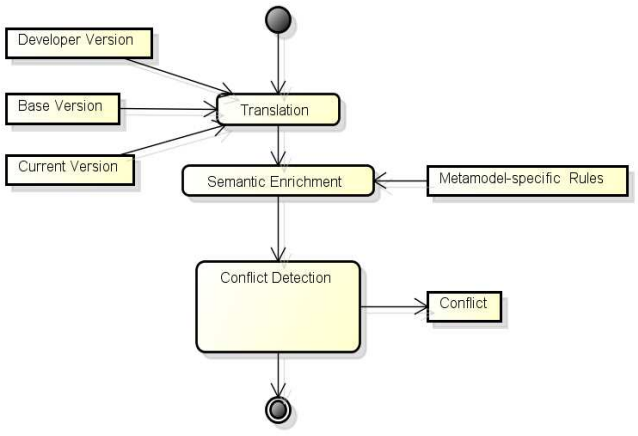



Figure 2. Our method overview

Where $v1$ and $v2$ can be any two variants in general, but in the specific case of this work, $v1$ = *current* and $v2$ = *developer*. Moreover, $\cap^*$ represents an especial intersection between sets that overloads the equality property to match not only identical elements. The matching between non-identical elements is possible because our method takes into account the syntactic rules of relationships described in the UCD metamodel. When a relationship is used, not only the relationship, but also the elements that compose it are verified. So, if an actor is deleted by $v1$, while $v2$ adds a relationship that uses the same actor (e.g. an association between this actor and a use case), a conflict is detected. This reasoning is analogous to the use of other relationships of the UCD metamodel.

The analysis of the $v1$, $v2$ and Conflict sets may lead to the following conclusions:

- If $v1 = v2$ then the versions are semantically equivalent. This means that the intentions of the developers were similar. In this case, there is no conflict and any one of the two versions can be chosen.
- If $(v1 \subset v2) \vee (v2 \subset v1)$ then one version semantically contains the other. In this case, there is also no occurrence of conflict and the intention of one developer entails the intention of the other. Since there is no divergence of intention, the most complete version should be chosen.
- If $(v1 \neq v2) \wedge \neg\big((v1 \subset v2) \vee (v2 \subset v1)\big) \wedge Conflict_{v1,v2} = \emptyset$ then the versions differ among themselves, one version semantically doesn´t contain the other but there is no semantic conflict. In this case, a syntactic merge suffices.
- If $Conflict_{v1,v2} \neq \emptyset$ then the versions semantically differ, one version semantically doesn´t contain the other and there are semantic conflicts. In this case, the Conflict set contains the syntactic facts that are implying semantic conflicts.

In order to have a better understanding of our method, section IV shows it in action.

## IV. OUR METHOD IN ACTION

Consider the UCD of a bank control system as depicted in Fig. 3, where Fig. 3.a shows the *base* version, Fig. 3.b shows the *current* version and Fig 3.c shows the *developer* version. These last two versions are revision of the *base* version.

The three versions have their files submitted to Translation activity, as explained in the previous section. The results produced in this activity are shown in Tab II. In this table, each column shows the automatically generated facts for each of the presented versions.

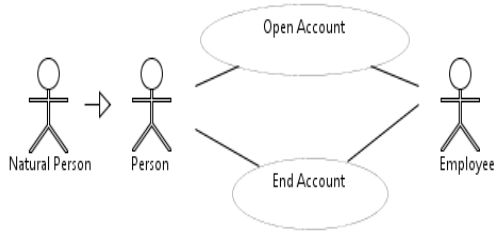
Then the Semantic Enrichment activity is performed through the application of previously defined semantic rules. These rules, written in Prolog, discover new facts created through the indirect relationships among model elements, as shown in Tab. III.

To illustrate the second activity of Fig. 2, consider the diagram presented in Fig. 3.a. Note that the Natural Person actor is indirectly associated with the Open Account use case. The association occurs through inheritance between Natural Person and Person actors. Considering the semantics of

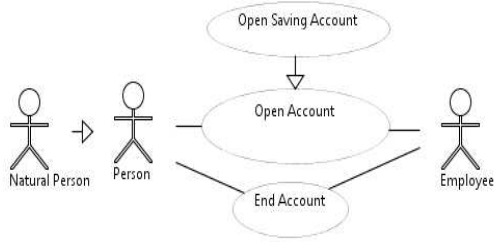
inheritance, it can be said that Natural Person is a Person. Furthermore, as Person is directly associated to the Open Account and End Account use cases, we can say that Natural Person is also associated with Open Account and End Account use cases, although the association is not directly shown in the model.

The discovery of the indirect relationships is a key factor in order to enable the detection of semantic conflicts in our method. The knowledge of those relationships allows us to verify that a change made to an element X generates conflict in an element Y, even if X and Y are not directly connected in the model.

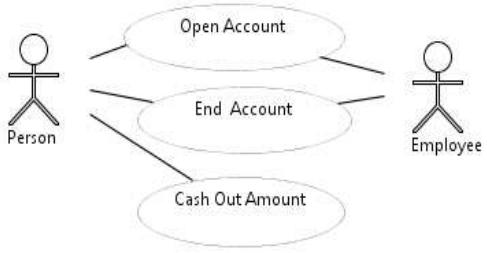
Tab. III shows the results of the second activity of Fig. 2 applied to the case shown in Fig. 3.




a. *base version*




b. *current version*




c. *developer version*

Figure 3. Model versions of a bank control system

The discovered facts are added to the files shown in Tab. II. Consider the *base* version column of Tab. III. The first Prolog fact of this column is association(natural_person, open_account). This fact can be visually inferred in Fig. 3.a where Natural Person actor is connected through inheritance with Person actor, which is associated with Open Account use case.

The third activity, Conflict Detection, computes the following sets:

$Del_{current} = \emptyset$

$Add_{current}$ = { usecase(open_saving_account),
inheritance(open_saving_account, open_account),
association(natural_person, open_saving_account),
association(person, open_saving_account),
association(employee, open_saving_account)}

$Del_{developer}$ = {actor(natural_person),
inheritance(natural_person, person),
association(natural_person, open_account),
association(natural_person, end_account)}

$Add_{developer}$ = { usecase(cash_out_amount),
association(person, cash_out_amount)}

Next, the intersection between the elements of the sets is calculated to detect occurrence of conflicts:

$Add_{current} \cap^* Del_{developer}$ = { actor(natural_person),
association(natural_person, open_saving_account) }

$Add_{developer} \cap^* Del_{current} = \emptyset$

$Conflict_{current, developer}$ = { actor(natural_person),
association(natural_person, open_saving_account)}

It can be observed that actor(natural_person) and association(natural_person, open_saving_account) operate over with natural_person actor. This intersection represents a conflict because an association requires the existence of both model elements in its association ends. According to Tab. II, natural_person is an actor. Moreover, according to $Del_{developer}$, this actor was deleted in *developer* version. Meanwhile, according to $Add_{current}$, a new association has been established in parallel with this actor in *current* version. This scenario shows that, in the same team, a developer is deleting an actor while another is expanded its responsibilities in the same system design. This situation indicates a semantic conflict between the versions. This type of conflict is not detected when only syntactic elements and direct relationships are analyzed, leading to false negatives.

As shown in the example, the presented method contributes to detect these conflicts by increasing the efficiency of the detect conflict method. Moreover, the computation of semantic equivalences identifies the use of different relationships that have the same meaning. This is

TABLE II. MODEL TO PROLOG TRANSLATION.

| Contents of prolog files – activity 1 | | |
|---|---|---|
| ***base* version** | ***current* version** | ***developer* version** |
| actor( person). | actor( person). | actor( person). |
| actor(natural_person). | actor(natural_person). | |
| actor(employee). | actor(employee). | actor(employee). |
| usecase(open_account). | usecase(open_account). | usecase(open_account). |
| usecase(end_account). | usecase(end_account). | usecase(end_account). |
| | usecase(open_saving_account). | |
| | | usecase(cash_out_amount). |
| inheritance(natural_person, person). | inheritance(natural_person, person). | |
| | inheritance(open_saving_account, open_account). | |
| association(person, open_account). | association(person, open_account). | association(person, open_account). |
| association(person, end_account). | association(person, end_account). | association(person, end_account). |
| association(employee, open_account). | association(employee, open_account). | association(employee, open_account). |
| association(employee, end_account). | association(employee, end_account). | association(employee, end_account). |
| | | association(person, cash_out_amount). |

TABLE III. SEMANTIC ENRICHMENT OF PROLOG FACTS

| Prolog facts added by semantic rules application | | |
|---|---|---|
| ***base* version** | ***current* version** | ***developer* version** |
| association(natural_ person,open_account). | association(natural_ person,open_account). | |
| association(natural_ person, end_account). | association(natural_ person, end_account). | |
| | association(natural_person, open_saving_account). | |
| | association(person, open_saving_account). | |
| | association(employee, open_saving_account). | |

possible because semantic rules abstract the syntactic differences of relationships and extract the meaning of such relationships in the model as a whole. Thus, the method does not identify these differences as conflicts and reduces conflicts false positives reported to developer. This feature reduces the rework generated for the team.

## V. PROTOTYPE IMPLEMENTATION

We are implementing a prototype of our method for UCD. Currently, the prototype entails activities related to comparison and conflict detection phases. We are testing these phases and we intend to extend our prototype in the near future to other UML diagrams.

For the purpose of testing our prototype, we adopt Papyrus [a] to design use case models. This tool provides an editing environment for EMF and UML models, among others. For each model, Papyrus generates two important files: a diagram interchange file, which contains the diagram information such as position of elements, and an XML Metadata Interchange (XMI) file, which contains the model elements themselves.

Each XMI file is submitted to Translation activity to be automatically transformed into a set of Prolog facts. The model´s transformation to Prolog facts is made using the OMG Model to Text (M2T) standard. The implementation of the transformation is based on Acceleo [b]. Acceleo is a generator that transforms models into code. It uses Model-driven Architecture (MDA) to transform a model into text. To perform the second activity, Semantic Enrichment, we adopt the TuProlog[c] library integrated with Java.

Our method was conceived to accommodate new types of diagrams and metamodels. As previously discussed, it is not restricted to proprietary model formats as input models are XMI files. The support for a new diagram or even a new metamodel requires three main tasks: writing a M2T transformation to generate Prolog facts according to the new diagram or metamodel, writing Prolog rules for the Semantic Enrichment phase and writing syntactic rules in Java to be used in Conflict Detection phase .

## VI. RELATED WORK

In [10], a semantic conflict detection method is presented, named Smover. The approach is based on semantic views of interest and inspection strategies of elements that can be configured by the user. A semantic view maps a metamodel to another based on relevant aspects of the first. The output of the transformation is a model in conformity with the second that contains the aspects of interest. The conflicts found in the original metamodel are syntactic conflicts, and those found the mapped metamodel are semantic conflicts. Our method transforms the elements and relationships into Prolog facts and uses inference rules to help compare two models. Moreover, we use only one metamodel to detect conflicts.

Odyssey-VCS [11] is a Model-driven VCS that allows the use of fine granularity for version UML 2 models. The conflict detection is based in existence analysis of elements and processing of attributes and relationships. It considers both non containment and containment relationships. Our method focuses in semantic conflict detection to all elements and relationships of the UCD models.

[a] http://www.eclipse.org/papyrus/

[b] http://www.eclipse.org/acceleo/

[c] http://tuprolog.alice.unibo.it/

Gerth et al. [12] present a method to detect conflicts that takes into account the semantics of business process models. This method decomposes the process model into fragments and activities to make your comparison. Moreover, it creates add, delete and move operations for fragments and activities. The approach also provides a method to the resolution of conflicts and uses individual strategies to resolve different types of conflicts. The method uses change-stated merge. Our work presents a method to state-based approach to UCD and in the future other UML models. We also intend to automatically resolve detected conflicts.

Koegel et al. [13] provide an algorithm to compute conflicts on the operations that change the model. It also takes into account the serialization of the application of these operations. The conflicts are classified into hard and soft. The hard conflicts must be resolved by the user and the soft ones automatically resolved. However, different of our method it does not take into account the semantics of the models and is made for operation-based approach.

Mirador [5] uses a hybrid state and operation-based approach. The merge is based on operation and detects direct and indirect conflicts. Conflicts are detected by the before(a,b) predicate where an operation a must come before an operation b. The approach describes techniques for detection and resolution of conflicts based on decision tables. The users can customize the rules of the tables. These tables can take into account the semantics and to use their rules to detect false positives. The approach uses metametamodel Ecore extended to compute differences between versions. Our method uses inference rules of metamodel to help compute semantic conflicts. It considers not only the false positive conflict detection generated by similar situations but also false negative conflict detection.

## VII. CONCLUSION

This paper presented a conflict detect method to MDE. The method expects three model versions as input (two variants with a common ancestry) and verifies: if the variants are semantically equivalent, if one variant semantically contains the other, and if there are semantic conflicts to be resolved. The process starts by transforming the models into Prolog facts. The Prolog facts are semantically enriched by means of metamodel-specific rules. Finally, semantic conflicts are discovered via three-way diff technique.

We also present an example that shows how changes in different elements can interfere with other elements, even if they are not directly connected. Due to the difficulty in identifying this type of conflict, the method helps on reducing the amount of false negatives conflict. Thus, it increases the efficacy of the conflict detection method as a whole.

Moreover, the detection of semantic equivalence decreases the amount of false positives conflicts reported to the developer, whereas purely syntactical analysis detects differences and reports them as conflicts. This feature also contributes to the improvement of conflict detection method because it reduces the rework of the team.

Currently, we are studying how to make the merge when there are no conflicts are detected. In the case of the equivalent models, the system must choose or suggest which model should be considered the merged version. To help on this suggestion, the traceability of indirect relationships should be considered. The traceability can indicate the best model designed.

As future work, we intend to support automatic conflict resolution and collaborative merge. At the phase of conflict resolution, heuristics may help on suggesting consistent solutions. Regarding collaborative merge, traceability can also figure as an important technique, visually guiding developers from semantic conflicts to the syntactic elements that triggered these conflicts.

We also intend to support additional UML diagrams and expand the method to work directly on the metametamodeling language, such as EMF and MOF, via the reflective API. This would allow processing any metamodel, requiring only the metamodel XMI file as input. Finally, we are planning to run some experimental studies with the proposed method.

## VIII. REFERENCES


[1] A. Cicchetti, F. Ciccozzi, e T. Leveque, “On the concurrent Versioning of Metamodels and Models: Challenges and possible Solutions”, 2011, p. 16–25.

[2] T. Mens, “A state-of-the-art survey on software merging”, *Software Engineering, IEEE Transactions on*, vol. 28, nº 5, p. 449–462, 2002.

[3] L. Murta, H. Oliveira, C. Dantas, L. G. Lopes, e C. Werner, “Odyssey-SCM: An integrated software configuration management infrastructure for UML models”, presented at the Science of Computer Programming, 2007, vol. 65, p. 249–274.

[4] K. Altmanninger, “Model Versioning – SMoVer”, *Smover: Configurable & Semantically Enhanced Conflict Detection in Model Version*, 2011. [Online]. Available: http://smover.tk.uni-linz.ac.at/prototype.php. [Accessed: 20-dez-2011].

[5] S. Barrett, P. Chalin, e G. Butler, “Table-driven detection and resolution of operation-based merge conflicts with mirador”, *Modelling Foundations and Applications*, p. 329–344, 2011.

[6] R. Conradi e B. Westfechtel, “Version Models for Software Configuration Management”, *ACM Computing Surveys (CSUR)*, vol. 30, nº 2, p. 232–282, 1998.

[7] J. G. Prudêncio, L. Murta, C. Werner, e R. da S. V. Cepêda, “To lock, or not to lock: That is the question.”, *Journal of Systems and Software*, vol. 85, nº 2, p. 277–289, 2012.

[8] K. Altmanninger e G. Kotsis, “Towards accurate conflict detection in a VCS for model artifacts: a comparison of two semantically enhanced approaches”, 2009, p. 139–146.

[9] K. Altmanninger, M. Seidl, e M. Wimmer, “A survey on model versioning approaches”, presented at the International Journal of Web Information Systems, 2009, vol. 5, p. 271–304.

[10] K. Altmanninger, W. Schwinger, e G. Kotsis, “Semantics for accurate conflict detection in smover specification detection and presentation by example”, *IJEIS*, p. 68–84, 2010.

[11] L. Murta, C. Corrêa, J. G. Prudêncio, e C. Werner, “Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System”, presented at the ACM, Leipzig, Germany New York, USA, 2008, p. 25–30.

[12] C. Gerth, J. M. Küster, M. Luckey, e G. Engels, “Detection and resolution of conflicting change operations in version management of process models”, *Software & Systems Modeling*, dez. 2011.

[13] M. Koegel, M. Herrmannsdoerfer,, e O. von Wesendonk, “Operation Base Conflict Detection”, presented at the IWMCP10: International Workshop on Model Comparison in Practice, Malaga, Spain, 2010.