Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Language-Specific Model Versioning Based on Signifiers

Philip Langer[a] Manuel Wimmer[b] Jeff Gray[c] Gerti Kappel[a]
Antonio Vallecillo[b]

a. Vienna University of Technology, Austria
b. Universidad de Málaga, Spain
c. University of Alabama, USA

Abstract In *model-driven engineering* (MDE), models constitute the central artifacts in the development process, and thus, are often built by teams of developers. As a consequence, adequate version control for models is crucial to the success of MDE-based projects. Several model versioning systems have been proposed recently. Most of them are generic in the sense that they are agnostic to modeling languages. Although this ensures a wide applicability, important merge issues may not be detected.

In this paper, we present an orthogonal extension to generic model versioning systems for enabling the detection of an important subset of language-specific merge issues. Users may enhance the versioning system's capabilities by defining *signifiers*, which describe the combination of features of a model element type that convey the superior meaning of its instances. Signifiers improve the different phases of the versioning process including comparing and merging models leading to a higher quality of the finally merged models. We showcase the applicability of our approach by enhancing the versioning support for the modeling language *Ecore*.

Keywords Model-driven Engineering; Model Evolution; Model Comparison; Model Versioning.

1 Introduction

Software systems are so large and complex that they have to be built by teams of developers [GJM02]. With the rise of model-driven engineering (MDE), *software models* are considered to be the main artifacts in the development process. Thus, tool support for building models in teams is highly needed [FR07]. For managing the team-based development of software, text-based versioning systems proved successful for code-centric software. However, applying such systems to the textual serializations of models (e.g., XMI [OMG07]) turned out to be unsatisfactory because of the impedance mismatch [NMBT05] between the text-based representation and the graph-based nature of models. Therefore, dedicated model versioning systems providing specific support for models have been proposed recently (cf. [BKL+12] for an overview).

Most model versioning systems are designed to be independent of specific modeling languages. Although this ensures a wide applicability, important merge issues may not be detected, because the specifics of modeling languages are neglected. Other systems are tailored to specific modeling languages, but they seem to be too inflexible, as they force developers to use for each modeling language its own versioning system.

Irrespective of being generic or specific, most model versioning systems rely on *artificial* universally unique identifiers (UUIDs) to determine whether two model elements in two successive versions of a model correspond to each other: if two elements share the same UUID, they are considered to be two versions of the same model element; otherwise, they are considered to be independent from each other, irrespective of their characteristics. However, there are versioning scenarios in which this simplistic strategy does not produce the optimal merge results: *(i)* two elements may share the *same UUID*, but they represent two *different entities* after concurrent changes, and *(ii)* two elements may have *different UUIDs*, but they actually represent the *same entity*; for instance, when two developers add similar elements concurrently. Using heuristic matching algorithms instead of UUIDs, these two drawbacks may be avoided, because the characteristics of the model elements are used for establishing model element correspondences. However, such heuristics may lead to imprecision of the match result causing other major drawbacks in certain versioning scenarios.

To combine the advantages of both approaches, i.e., UUIDs and natural model element characteristics, we present an orthogonal lightweight extension to generic model versioning systems for enabling the detection of an important subset of language-specific merge issues. Therefore, users may enhance the versioning system's capabilities by specifying so-called *signifiers*, which describe the combination of specific features of a model element type that are paramountly decisive for the meaning of its instances. Instead of using artificial UUIDs only, signifiers are applied to explicate and regard the natural identifier of a model element to eliminate the two aforementioned shortcomings of pure UUID-based approaches.

In this paper, we introduce the notion of signifiers and their usage in model versioning to deal with an important subset of language-specific merge issues. In particular, we show how signifiers can be realized based on the Epsilon Comparison Language [Kol09, KRP11] and demonstrate their integration in the model versioning process in order to achieve merged models of a superior quality. We demonstrate the usefulness of our approach by showcasing how the versioning support for the modeling language Ecore [SBPM08] is improved and discuss our experiences with our prototypical implementation of the signifier-based versioning mechanisms, which has been integrated in the model versioning system AMOR[1] [BKS+10].

The remainder of this paper is structured as follows. In Section 2, we present the state of the art in model versioning and introduce a versioning scenario in Section 3, which serves as a running example throughout the paper. The notion of signifiers and how they are specified is discussed in Section 4. In Section 5, we introduce the integration of signifiers in the model versioning process. Subsequently, we present a case-based evaluation and a critical discussion of the approach in Section 6, before we conclude the paper with an outlook on future work in Section 7.

[1]`http://www.modelversioning.org`

2 State of the Art

Existing model versioning systems may be categorized based on two orthogonal dimensions. First, they differ regarding their approach to obtain the changes applied concurrently to the common ancestor version: the changes may be either *directly recorded* in the modeling editor on-the-fly or obtained using *model comparison* afterwards. Second, model versioning systems may either be tailored to a certain modeling language or they are generic; that is, the degree to which they use generic algorithms that exploit reflection mechanisms to access the models under consideration. Thereby, the model versioning system is independent of the respective modeling language. In the following, we discuss these two dimensions in more detail.

2.1 Change Recording versus Model Comparison

In the context of model versioning, obtaining the changes that have been applied between two succeeding versions of a model is a crucial prerequisite for computing conflicts among the identified changes and creating a merged model. The most straightforward way to obtain the applied changes is to record them as they are performed in the modeling editor. This technique is often referred to as *operation-based versioning* [LvO92] and is used in the model versioning systems CoObRA [SZN04], Oda & Saeki [OS05], and EMFStore [KHWH10]. Obviously, such approaches depend heavily on the modeling editor that is used by developers to create and modify the models, because the editor has to capture and provide all occurring edit events in a processable format. Nevertheless, change tracking allows for very precise change logs.

An alternative to change recording is to obtain the changes using model comparison approaches. Such approaches, which are often referred to as *state-based versioning*, do not record the changes directly, but compute them from the original and two revised *states* of a model. Model comparison is applied by several model versioning systems, such as proposed by Alanen & Porres [AP03], Mehra et al.[MGH05], Oliveira et al. [OMW05, MCPW08], EMF Compare [BP08], and Gerth et al. [GKLE10].

Model comparison is usually realized in two phases, a matching phase and a differencing phase. The matching phase aims at establishing one-to-one correspondences between model elements in the original model and the revised models. Based on the established match, the differencing phase computes the actual differences (i.e., the changes) between all corresponding model elements. Whereas the differencing phase is very similar from a conceptual point of view among all approaches, the matching phase differs concerning the used match function. The match function may either rely on artificial UUIDs that are attached to each model element, or the match function adopts similarity-based heuristics to establish the correspondences among the model elements (cf. [Kol09] for a survey on model matching). UUID-based model matching is used by Alanen & Porres [AP03], Mehra et al.[MGH05], Murta et al. [MCPW08], and Gerth et al. [GKLE10]. EMF Compare [BP08] and the approach by Rivera & Vallecillo [RV08] allow to use either UUID-based or similarity-based matching. There are several other tools that focus on similarity-based model differencing only, such as UMLDiff [XS05], DSMDiff [LGJ07], and SiDiff [SG08].

The accuracy of the model matching technique has a direct impact on the accuracy of the model difference. For instance, if an updated model element in the revised model could not be matched with its corresponding original model element, the model comparison reports that the model element has been deleted instead of reporting the update. Thus, most of the model versioning tools that are based on model comparison use UUIDs to allow for finding corresponding model elements, even if the model element has been changed significantly in the course of a revision. As we will see later, this approach also has major drawbacks in several model versioning scenarios.

2.2 Generic versus Language-specific Model versioning

Existing approaches for model versioning are either designed to work independently of the modeling language (i.e., they are generic) or they are tailored to a specific modeling language.

Most of the existing *generic* model versioning systems make use of the reflection mechanisms of the underlying metamodeling framework (e.g., Eclipse Modeling Framework (EMF) [SBPM08]). Thus, all modeling languages defined by the underlying metamodeling framework are handled uniformly. Such generic versioning systems have been proposed by Mehra et al. [MGH05], Oda & Saeki [OS05], Murta et al. [MCPW08], EMF Compare [BP08], and EMFStore [KHWH10].

The major drawback of generic model versioning systems is that they neglect the specifics of the modeling language to which the models under version control conform. In Section 3, we illustrate a versioning scenario in which language-specific knowledge helps significantly to achieve a merge result of higher quality.

Besides generic model versioning systems, other approaches have been proposed, which are tailored especially for a specific modeling language. The specifics of a modeling language is encoded across all phases of the merge process in order to improve the accuracy of the change and conflict detection, as well as the quality of the merge result. Thereby, language-specific changes and conflicts may be detected and dedicated merge rules may be applied. Language-specific versioning systems have been presented by Schneider et al. [SZN04] and Gerth et al. [GKLE10]. However, such systems are very inflexible, because users have to employ their own versioning system for each modeling language they use.

Another idea to address language-specific conflicts has been presented by Cicchetti et al. [CDRP08]. Using their approach, conflicts can be specified manually in terms of conflict patterns. These conflict patterns are represented by difference elements, which are reported as conflict whenever found in the combination of two difference models. To this end, a hand-crafted set of language-specific conflict patterns can be established to create a dedicated conflict detection system. Thereby, the realization of a customizable conflict detection component is possible. Their approach also allows to specify reconciliation strategies to specific conflict patterns. It seems to be a great deal of work to establish a complete set of conflict patterns for a specific language, as all potentially conflicting changes have to be specified explicitly; nevertheless, a highly customized conflict detection tool can be achieved in the end.

To combine the advantages of generic and language-specific model versioning systems, we recently developed the *adaptable model versioning system* AMOR [BKS$^+$10]. AMOR offers generic versioning capabilities out-of-the-box, but it is adaptable to enhance the versioning support for a specific modeling language. Thereby, developers are empowered to balance flexibly between reasonable adaptation efforts and the required level for versioning support. In the remainder of this paper, we present one particular adaptation point of AMOR. The general approach presented in the following, however, is not limited to be used within AMOR, as it can be integrated easily with any other model versioning system.

3 Motivating Example

In this section, we discuss an excerpt of a model versioning scenario (cf. Figure 1) that motivated us to perform the work presented in this paper. In the original version V_o, there is an Ecore model consisting of three classes, namely Developer, Manager, and BugReport. The class Manager contains a reference named createdReports, which refers to the class BugReport. The gray circles in Figure 1 indicate the UUIDs of the respective model elements; e.g., the class Developer has the UUID `c1`.

This original version is modified concurrently by two developers. Developer 1 performs the modification m_1 to obtain the revised version V_{r1} and developer 2 performs m_2 leading to V_{r2}.

Developer 1 addresses the requirement that instances of both the class Manager and the class Developer should be able to refer to bug reports using the reference createdReports. Therefore, she moves the reference createdReports from the class Manager to Developer and specifies the class Developer to be the superclass of Manager. Additionally, she adds a new reference called assignee with a multiplicity of `1..1`, as well as an attribute named id to the class BugReport.

Developer 2 is concerned with the task of allowing instances of the class Manager to refer to all developers that are managed by the respective manager. Thus, a reference between Manager and Developer is required. As frequently happening in practice, she copies the existing reference createdReports instead of creating a new reference, because the copied reference is very similar to the reference she intends to create. As it is not easily possible to distinguish the copies from each other in the modeling editor, she modifies the *original reference* having the UUID `r1` according to her needs instead of changing the created copy having the UUID `r3`. More precisely, she sets the target of the original reference `r1` to Developer and changes its name to managedDevelopers. Besides these changes, she renames the class BugReport to IssueReport and also adds a new reference named assignee, which links IssueReport and Developer.

Let us now consider the merge results of present state-of-the-art model versioning systems for the aforementioned versioning scenario. As discussed in the previous section, we may distinguish between operation-based approaches and state-based approaches. State-based approaches may further be categorized according to their approach for matching two revisions of a model; that is, the match is either established based on synthetic UUIDs or on similarity-based heuristics. The merge result

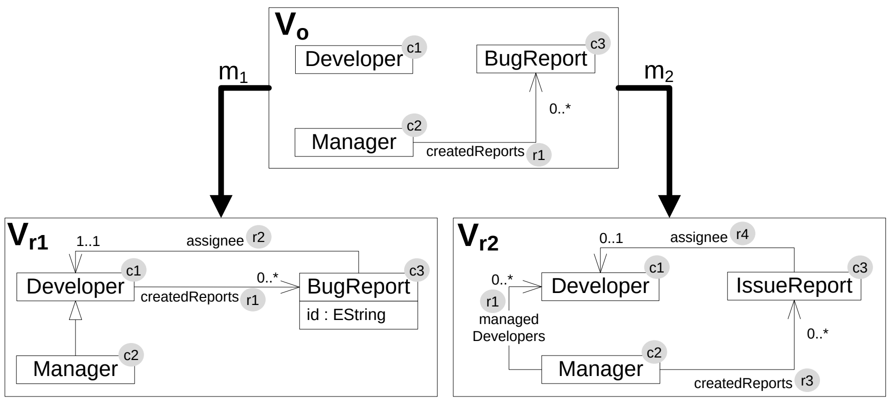
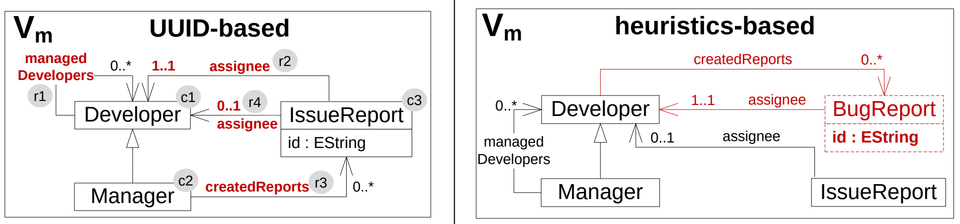


Figure 1 – Motivating Model Versioning Example



(a) Merged Version using Operation-based or state-based Merge using UUIDs

(b) Merged Version using State-based Merge without UUIDs

Figure 2 – Merge Result of Current Model Versioning Systems

mainly depends on whether or not the model versioning system captures and exploits synthetic identities of model elements during the merge construction; this is true for operation-based approaches (because they use direct object references), as well as for state-based approaches working with UUIDs. It is not the case, however, for state-based approaches using similarity-based heuristics for model matching. Hence, we investigate two different merge results for our versioning scenario in the following: the one illustrates the result when synthetic identities of model elements are used (cf. Figure 2a), and the other one arises from using similarity-based heuristics (cf. Figure 2b). Please note that these results may differ slightly from the actual results computed by the respective model versioning systems (e.g., due to different heuristic match functions). However, in this context, we focus on the results to be *expected* from the different underlying *concepts* of UUID-based and heuristics-based merging approaches.

Merge Result based on UUIDs. When merging two revisions using an operation-based or UUID-based merge approach, the identities of the model elements are preserved, irrespectively of how they have been modified in both revisions. When applying such a merge to our example depicted in Figure 1, no conflict is reported, because no overlapping changes have been applied in m_1 and m_2. Thus, all changes of both revisions can be unified, which leads to the merged model V_m in Figure 2a. However, three issues are introduced in the merged model.

The first issue concerns the reference with the UUID r1. According to the modifications in m_1, this reference has been moved from the class Manager to Developer. However, the same reference has been subject to change also in m_2. In particular, it has been renamed from createdReports to managedDevelopers and its target has been changed from BugReport to Developer. As a result of applying all changes of m_1 and m_2 to this reference, it now has been turned into a reflexive reference in the class Developer. Obviously, this is neither what developer 1 wanted, nor what developer 2 intended the reference to express. However, no conflict is reported, because both changes affected *different* properties of the reference and, as a result, they can be applied without interfering with each other. A generic merge approach is not aware that a concurrent change of a reference's source and target might obfuscate the intended meaning of the reference.

The second issue arises from the concurrent insertion of the equally named reference called assignee. When applying all changes of both modifications m_1 and m_2, the reference with the UUIDs r2 and r4 are both added to the class IssueReport. Thereby, the merged model exhibits an unfavorable redundancy, as well as a validation error: the modeling language Ecore forbids equally named references in the same class, because this would lead to a compile error in the generated Java code. A generic merge approach is designed to be agnostic of the respective modeling language and, as a consequence, is not aware of redundancies and language-specific validation constraints. This is even more aggravated by the fact that the two model elements are not entirely equal: the lower bound of the references are different.

Finally, as a consequence of the flawed merge of the reference with the UUID `r1`, also the reference createdReports with the UUID `r3`, which is the copy of the original reference named createdReports created by developer 2, contradicts the intention of developer 1. Developer 1 intended to move this reference from Manager to Developer; and developer 2 did not actually intend to change the reference actively. In the merged model, however, the reference still resides in the class Manager instead of in the class Developer, because the merge approach assumes that the reference has been newly created by developer 2 in the class Manager since the reference's UUID does not match with any UUID in the original model.

On the upside of the UUID-based merge, no conflict has to be resolved and all changes could be merged easily because of the fine-grained and precise change detection of such approaches. Even the class with the UUID `c3`, which has been renamed in one revision, used as a reference target of the new reference createdReports, and extended with an additional attribute (named id) could be merged so that all original changes applied to this class are reflected.

Merge result based on heuristics. Please note that the merge result may vary based on the applied heuristics of the respective approach. In the following, we discuss the most likely results that are obtained from using state-of-the-art tools, such as EMF Compare [BP08], but also discuss possible variations of the results coming from different heuristics. A potential merge result that is obtained from applying a heuristics-based merge approach is depicted in Figure 2b. As the identity of model elements is not preserved based on synthetic UUIDs or object references, the issue concerning the reference createdReports, which has been moved to the class Developer in m_1 and renamed to managedDevelopers in m_2, does not occur. Heuristics-based merge approaches are usually not able to match the reference createdReports in V_o with the reference managedDevelopers in V_{r2}. Instead, the reference in V_o is matched with the equally named corresponding references in V_{r1} and V_{r2}. Thus, the merge algorithm assumes that the reference has not been changed in the course of the modification m_2 and that it has been moved to the class Developer in V_{r1}. Consequently, the reference is also moved to Developer in the merged version, which is an improvement over the situation with UUID-based merge approaches. However, we get other severe merge issues with heuristics-based approaches.

First, the merge most likely reports several "unnecessary" conflicts (i.e., *false-positive* conflicts), because of the class BugReport. As this class has been renamed in V_{r2}, the original version of this class in V_o could probably not be matched with the renamed version of it in V_{r2}. Thus, the algorithm assumes that the class has been deleted in V_{r2} and a new class named IssueReport has been added. However, the class BugReport has been modified in the opposite revision V_{r1}; as a result, a conflict is raised because the merge algorithm cannot apply both the deletion of BugReport and the insertion of the attribute id, the insertion of the reference assignee, as well as setting the target of createdReports to the deleted class BugReport. Second, the redundancy concerning the reference assignee, which has been added concurrently, still remains, because currently existing approaches do not merge concurrently added

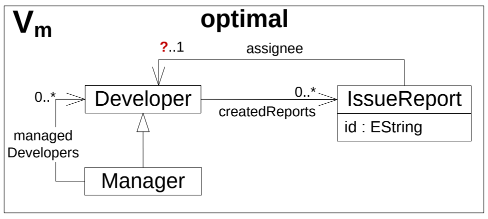


Figure 3 – Optimal Merge

model elements, irrespectively of whether they would match according to the applied heuristics. The matching algorithm might also miss matching the reference createdReports in V_o with the moved reference in V_{r1}, depending on the applied heuristics; in this case the result would even be worse, because we would have another reference called createdReports in the class Manager referring to IssueReport.

Optimal merge result. One potential merged model, which better reflects the intentions of both developers, is depicted in Figure 3.

In this merged model V_m, the original reference createdReports has been split: one reference reflects the intention of developer 1 (i.e., the reference createdReports between Developer and IssueReport) and the other one reflects the intention of developer 2 (i.e., the reference managedDeveloper between Manager and Developer). Admittedly, this is only *one possible* way of resolving this issue. In any case, the developers should be confronted with a warning to point their attention towards their indirectly contradicting changes regarding the meaning of the reference that has been concurrently changed. Unfortunately, current model versioning systems are not able to identify such merge issues, because they are unaware of the fact that a concurrent change of a reference's source and target may obfuscate its meaning and that concurrent changes of such meaningful properties of a model element should be treated specifically.

The duplicate reference assignee has been merged into one reference to avoid the redundancy. However, the references are not entirely equal (cf. lower bound of assignee). Thus, an ideal model versioning system should raise a warning for indicating the need for a decision concerning the lower bound of the reference in the merged model. Current model versioning systems are not capable of identifying such redundancies in the merged model, especially if these systems follow a generic approach and if the redundant model elements are not entirely equal. To allow for detecting such unfavorable redundancies, additional information is needed to enable a model versioning system to decide whether two model elements are redundant or not. In our example, it would have to be aware of the fact that two references with the same name and the same source are redundant.

Discussion. With this example, we illustrated the advantages and disadvantages of UUID-based and heuristics-based merging and showcased the insufficiencies of both. In general, UUID-based merging is more precise and leads to less conflicts, as well as a better merge result. However, it might also lead to undetected obfuscated meanings of model elements. Such a scenario usually occurs if the meaning of a model element has been concurrently modified in a contradicting way, as was the case with the reference with the UUID `r1` in our example discussed above. Furthermore, UUID-based merging fails to detect overlapping meaning of different model elements.

Heuristics-based merge approaches do not exhibit this shortcoming. If a model element is changed significantly by two developers in the course of two concurrent revisions, the model element is implicitly split, because it could probably not be matched anymore based on heuristics. Thus, the model versioning system treats the concurrently modified model element as two independently created model elements. As a result, the concurrent changes are not merged into one model element so that the meaning of the original model element is not obfuscated during the merge. However, if a model element could not be matched due to significant changes, the model versioning system assumes that the model element has been deleted. As we have seen in our example, this might again lead to false-positive conflicts as was the case with the class BugReport. Moreover, concurrently added yet very similar model elements are not merged into one model element in the final version by current approaches. Thus, irrespective of the applied heuristics, we would still end up with redundancies in such scenarios as it happened with the reference assignee.

Thus, we argue for an approach that combines the advantages of both techniques. More precisely, we advocate the use of artificial UUIDs to achieve a precise match, but also aim at treating the properties carrying the superior meaning of model elements specifically during the merge.

4 Signifiers of Model Elements

The goal of the research described in this paper is to extend generic model versioning systems so that they are capable of detecting inadvertently obfuscated and unexpectedly overlapping meanings of model elements as illustrated in the previous section. Therefore, we have to find a way of specifying the meaning of model elements first.

4.1 Meaning of Model Elements

Irrespective of whether a model element is a linguistic or an ontological instance of a metaclass [AK03, Küh06], it always represents a *referent*[2]. Hence, we may define the *meaning* of a model element as the *relationship* between the model element and its represented referent. This relationship, however, is imputed and potentially ambiguous; it is established through a thought of the referent in the subjective mind of the person perceiving the model element. For clarifying the relationship between a model element, the thought of the referent, and the referent itself, we adopt the idea of the *semiotic triangle* [OR23], which was designed originally to describe the relationships among thoughts, words (of a natural language), and things.

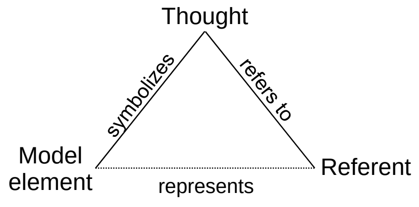


Figure 4 – Semiotic Triangle of Model Elements

[2]According to Merriam Webster, referent is defined as the thing that a symbol (as a word or a sign) stands for.

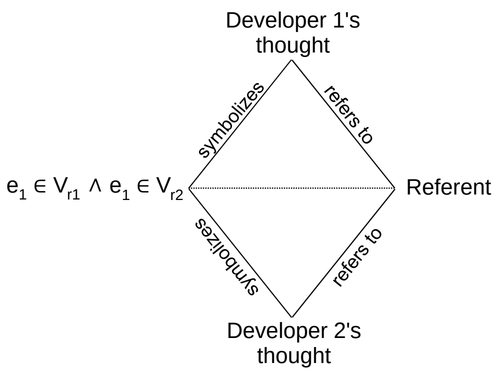
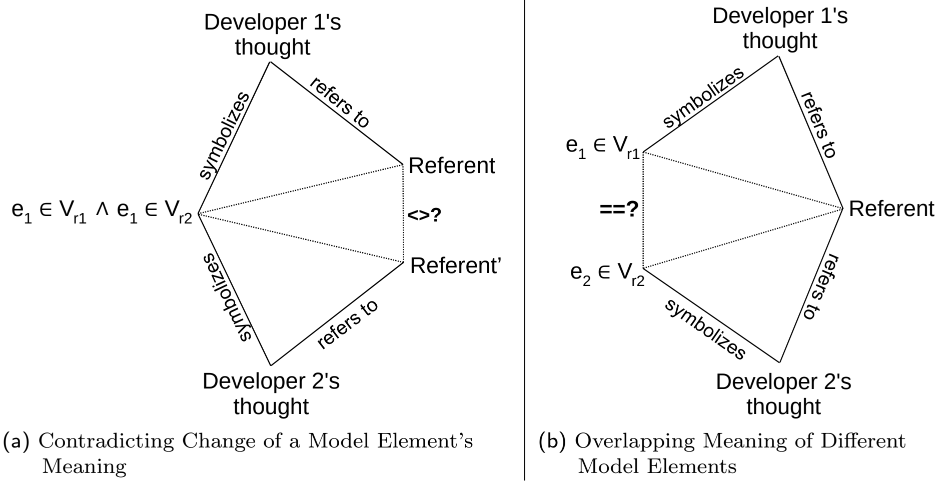


Figure 5 – Semiotic Triangle in Multi-user Settings: Ideal Case

The adoption of the semiotic triangle for the meaning of model elements is depicted in Figure 4. The corners of this triangle are constituted by a *referent*, the developer's *thought* of this referent, and a *model element* standing for the referent. Whenever a developer intends to model a thing (i.e., the referent), it evokes the developer's thought, which *refers to* the referent in the developers mind. By *symbolizing* this thought using a concept of the respective modeling language, the developer creates a model element in order to *represent* the referent (cf. Figure 4).

In the context of model versioning, the same model may now be modified concurrently. Thus, a model element created by one developer evokes a thought of another developer. In an ideal case, this thought refers to the same referent; that is, the model element *represents* the same thing for both developers (cf. Figure 5). Thus, the developers would usually not modify the model element contradictorily concerning its intended meaning. However, there are two problematic cases in opposition to this ideal case, which are depicted in Figure 6.



(a) Contradicting Change of a Model Element's Meaning

(b) Overlapping Meaning of Different Model Elements

Figure 6 – Diverging Semiotic Triangles

Contradicting change of a model element's meaning. The first problematic case concerns the concurrent *change of the meaning* of the *same* model element (cf. Figure 6a). A model element e_1 is affected in the course of two concurrent revisions V_{r1} and V_{r2}. Assuming the synthetic identity in terms of a UUID or object reference of this element is preserved, the model element e_1 is still part of both revised models V_{r1} and V_{r2}. In the concurrent revisions, however, the same model element e_1 now represents *different* referents according to the thoughts of the involved developers. To detect such scenarios, a model versioning system would have to be aware of which concurrently changed characteristics of a model element cause the represented referents to diverge and what differences between the model element e_1 in V_{r1} and the model element e_1 in V_{r2} make up the diverging interpretations of the model elements (cf. `<>?` in Figure 6a). In Section 3, we illustrated such a scenario with the concurrent change of the reference createdReports with the UUID `r1`.

Overlapping meaning of different model elements. We may also encounter the opposite of the previous case. More precisely, an issue arises if *two distinct model elements*, e_1 and e_2, represent the *same referent* in the revised versions V_{r1} and V_{r2}, respectively (cf. Figure 6b). In such a case, we end up with an unfavorable redundancy in the merged model, if the overlapping meaning remains undetected. As opposed to the previous case, a model versioning system would have to be aware of which characteristics of two model elements cause an overlapping interpretation (cf. `==?` in Figure 6b). This seems to be possible without additional knowledge on the meaning of model elements, if both model elements are entirely equal. However, model elements having different characteristics may still represent the same referent. Moreover, not only the model element itself, but also its context, is an important factor for conceiving its meaning, which again impedes the detection of such a scenario. A simple example of such a scenario has been demonstrated using the reference assignee in Section 3.

4.2 Signifiers of Model Elements

The knowledge on which characteristics of a model element are paramountly decisive for its meaning is certainly specific to the modeling language the model element conforms to. Thus, for enabling generic model versioning systems to detect concurrent changes of the meaning of the same model element or the overlapping meaning of distinct model elements, we have to specify the characteristics of model elements that are of major importance for its meaning. Therefore, we adopt the notion of signs and signifiers in linguistics [DS16] and introduce the term *signifier* to refer to characteristics in terms of one or more intrinsic or extrinsic properties of a model element that are paramountly decisive for conveying its meaning. Thus, signifiers explicate the "shape" of a model element pointing primarily to the represented referent. For instance, the meaning of a reference in Ecore is mainly conveyed by its containing class, its name, and its target; thus, the signifier of a reference in Ecore is a combination of its containing class, its name, and its target. The properties constituting the *signifier* of a model element may be thought of as the *natural identifier* of the model element. Please note that a model element's signifier does not consist only of *one intrinsic* property of the model element; it usually is a *combination of multiple properties*, which may also come from its context, such as its child model elements, its container, or cross-referenced model elements. As these properties constituting the signifier are particularly important for the meaning of a model element, we argue that they should be treated specifically when merging concurrent changes of a model.

Specification of signifiers. A signifier specification for a model element type has to state *which features* have to be considered for computing a model element's signifier and *how to combine* the values of these features. Ideally, this specification should be *directly applicable* to examine whether two model elements share the same signifier or not. One powerful technology that fulfills these requirements is the Epsilon Comparison Language[3] (ECL) [Kol09, KRP11]. Originally designed for model matching, ECL provides a domain-specific language for developing language-specific model comparison rules. As ECL is based on the Epsilon Object Language [KRP11], also imperative programming statements, as well as plain Java libraries may be incorporated into ECL rules. Thus, we decided to use ECL as the base technology for specifying signifiers and detecting signifier-based merge issues. Please note, however, that ECL is only one possibility to realize signifiers and their usage in model versioning. Before we proceed with presenting our approach for detecting signifier-based merge issues, we give a brief overview on ECL in general and discuss how ECL can be used for specifying signifiers.

```
1  // Signifier specification for Ecore references
2  rule EReference2EReference
3    match left : Left!EReference
4    with right : Right!EReference {
5    // two references, referred to as left and right, share the same signifier if ...
6    compare :
7      // their names are equivalent,
8      left.name = right.name and
9      // their sources are equivalent,
10     left.eContainer.matches(right.eContainer) and
11     // and their targets are equivalent
12     left.eType.matches(right.eType)
13 }
14 ...
```

Listing 1 – Example of a Signifier Specification for `EReference`

In general, each ECL rule declares a name and two parameters, let us call them `left` and `right`, that specify the types of elements the rule can compare [KRP11]. Additionally, such a rule contains a `compare` block specifying declaratively the match condition indicating whether or not the two elements should correspond to each other. Within the `compare` block, users may define the actual comparison logic by accessing feature values of the two elements, applying comparison operators to these values, or by arbitrary expressions returning a boolean value. Several expressions may again be combined using logical operators (cf. [Kol09, KRP11] for more information on ECL).

For specifying the signifier of a model element type, we may use one or more ECL rules, which compare instances of the respective type with each other. If the rule indicates a match for two model elements (i.e., the `compare` block is fulfilled), the model elements share the same signifier, and consequently, represent the same referent assumingly.

An example of such an ECL rule is given in Listing 1, which specifies the signifier of Ecore references. As a signifier specification rule compares two model elements of the *same type* with each other, the head of the rule is specified for `Left!EReference` to be matched with `Right!EReference` in lines 3 and 4 of Listing 1. Thereby, the rule is specified to be applicable to instances of `EReference` from the model `Left`

[3]`http://www.eclipse.org/gmt/epsilon/doc/ecl`

and instances of `EReference` from the model `Right`. Moreover, the two instances are assigned to the local variables `left` and `right`, respectively, which may now be used in the `compare` block of the rule. In this `compare` block, a signifier specification typically contains expressions comparing the natural identifiers, such as the `name` of a reference (cf. line 8 in Listing 1). However, in several cases, *multiple features* of a model element convey its meaning *in combination*. Therefore, we may combine multiple expressions, each comparing one feature, in terms of conjunctions in an ECL rule. This has been done to further include the reference source and the reference target into the signifier computation of Ecore references in lines 10 and line 12, respectively. In Ecore, the source of a reference is constituted by its containing class (i.e., `eContainer`) and the target is represented by the feature `eType`. Please note that we use an operation called `matches` for comparing the source and target class of the reference, in contrast to the equals operator (`=`) that we have used for the reference name. Whereas comparison operators, such as `=` and `>`, compare the specified values *directly*, the `matches` operation, which is built into ECL natively, delegates the comparison to another comparison rule. Besides the advantages of decomposing the comparison logic into multiple rules, using the `matches` operation is also necessary in this case, because the compared model elements reside in different models, `Left` and `Right`. Thus, the expression `left.eContainer = right.eContainer` would never be true unless the container of `left` and `right` is the *exact same model element*. Consequently, the common comparison operators (e.g., `=`, `>`, etc.) should only be used for simple data types.

5 Detection of Merge Issues based on Signifiers

In this section, we present our approach for detecting language-specific merge issues based on signifiers. This approach is intended as an orthogonal extension for generic model versioning systems. Thus, we first discuss the applied overall versioning process to set the context to subsequently introduce two signifier-based conflict detection mechanisms: one for revealing contradicting changes of the meaning of one model element and one for detecting an unexpected overlapping meaning of two model elements.

5.1 Versioning Process

To clarify the context of the proposed signifier-based merge issue detection, we depict an overview of the model versioning process of AMOR in Figure 7. Please note that the presented approach is not limited to be used within AMOR. On the contrary, it may be integrated with any other model versioning system.

The input of the versioning process are three models: the common original model V_o and two concurrently changed models, V_{r1} and V_{r2}. Thus, V_{r1} is the result of the first modification m_1 performed by developer 1 and V_{r2} is the result of m_2 performed by developer 2.

As the AMOR versioning process applies state-based model comparison, the first step is *model matching*. In this step, the revised models are each *matched* with the common original model V_o. From that, two *match models*, called $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$, are obtained, which mark the corresponding model elements among the original model V_o and the revised models V_{r1} and V_{r2}, respectively.

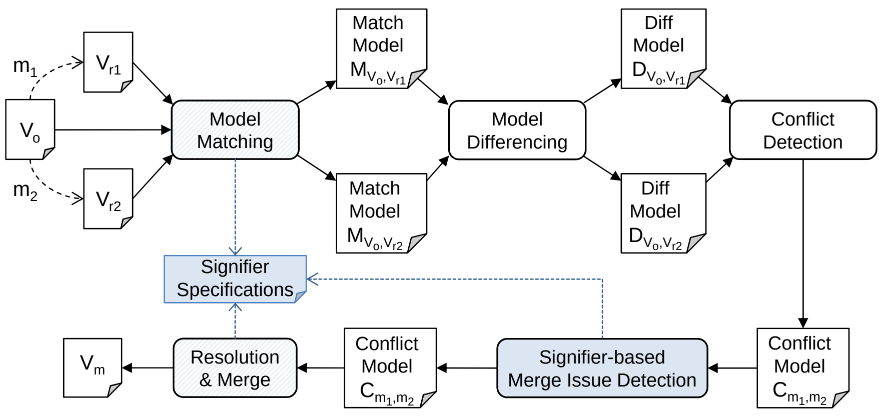


Figure 7 – Signifier-enhanced Versioning Process

Based on these match models, in the next step, the actual *differences* between the original version and the revised versions are computed. More precisely, for each pair of corresponding model elements, a fine-grained comparison is performed. If differences between the two model elements are encountered, a description of the difference is added to the respective difference report, called $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. Alternatively, model versioning systems may apply change recording [HK10, LvO92] to establish the difference models.

The two difference models are the input of the *conflict detection*, which analyzes the concurrently applied changes in $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$ to compute conflicting pairs among them. A pair of concurrent changes is marked as conflicting if they do not commute (also referred to as *update-update conflict*) or if one change cannot be applied after the opposite change has been applied (e.g., a *delete-update conflict*). Change-based conflict detection is, however, not the focus of this paper. For more information on such conflict detection mechanisms, we kindly refer to [AP03, KHWH10, TELW12, Wes10]. All detected conflicts are added to a conflict model, called C_{m_1,m_2} in Figure 7, which is passed on to the next step.

Before the conflicts in the conflict model are resolved and a merged version of both revisions is created, we integrate the *signifier-based merge issue detection* proposed in this paper. Signifiers are specific to the modeling language of the models under version control. Thus, this step in the versioning process relies on pre-specified signifier specifications in terms of ECL rules, which have to be created for each modeling language being used. Based on these specifications, the versioning system may reveal signifier-based merge issues as discussed in the remainder of this section. If signifier-based merge issues are detected for a certain versioning scenario, a description of the respective issue is added to the conflict model.

Finally, all raised conflicts are resolved by the developers so that a merged version of the model, called V_m in Figure 7, can be created in the last step.

Improving model matching using signifiers. The primary focus of this paper is the detection of signifier-based merge issues and how such raised issues may improve the final merge result. However, the signifier specifications may also be used to improve the quality of the match models and, as a result, the accuracy of the model differencing, as discussed in the following.

Although UUID-based matching is probably the most efficient and straightforward technique for obtaining the actual model changes, there are some drawbacks of this approach. In particular, if model elements loose their synthetic identity (constituted by the UUID or by an object reference), it is assumed in the model differencing step that the model element has been deleted and a new model element has been added in the considered revision. Thus, concurrent changes to the same model element in a concurrent revision cannot be merged and will always cause a conflict (i.e., a *delete-update conflict*). Unfortunately, such a scenario occurs quite frequently; not only because the developer deletes and re-creates the same model element subsequently, but also because of improperly implemented copy & paste or move actions in certain modeling editors, which fail to retain the UUID of the involved model element (e.g., in the tree-based Ecore editor).

To address these drawbacks, we apply a two-step matching process: first, a UUID-based matching is applied to obtain a base match, which is improved subsequently by applying signifier match rules to the pairs of model elements that could not be matched based on UUIDs. Thereby, the advantages of UUID-based matching are retained and its drawbacks are reduced significantly. As the comparatively slow signifier-based matching is kept at a minimum with this hybrid approach, the execution time of the model matching phase should still be reasonable (please see Section 6 for a performance analysis).

5.2 Integrating Match Models in Signifier Specifications

After the conflicts among the detected differences have been revealed, we may start with detecting signifier-based merge issues. Therefore, we apply the signifier specifications to compute the signifiers of certain model elements and check for contradicting *signifier changes* of the *same* model elements, as well as for *equal signifiers* of *different* model elements. Whether two model elements are considered to be the same or different has to be determined based on the *match model* that has been computed in a previous step of the versioning process.

However, ECL does not provide a method to access previously computed match models from ECL rules directly. For comparing referenced model elements from within a rule, ECL only provides the built-in operation called `matches` (as used in Listing 1). The goal of this operation is to refrain from duplicating the comparison logic and to de-couple match rules from each other. When this operation is called, the ECL engine searches for an appropriate match rule based on the referenced model elements' type, applies the found match rule, and returns the resulting boolean value. If no appropriate match rule is found, it returns `false`.

Whereas this functionality is perfectly suitable for model matching in general, it does not allow us to check whether two model elements are the same according to the match model computed in the previous step of the versioning process. Let us consider the signifier specification for `EReference` in Listing 1 as an example. The signifier of Ecore references is computed based on a reference's name, its source, and its target. More precisely, if two references have the same name, *the same source class*, and *the same target class*, they share a common signifier. Thus, the signifier specification should return true if the source class of a reference in the original model V_o, for instance, is considered to be the same source class in the revised model V_{r1}; that is, if there is a correspondence between the source class in V_o and in V_{r1} in the match model $M_{V_o,V_{r1}}$ (cf. Figure 7). In the signifier specification for `EReference` in Listing 1, the operation `matches` is applied to compare the source class (and the target class) of a reference. However, the comparison is delegated to other match rules, which are in fact again signifier specifications for the type `EClass`. As a result, this rule compares the *signifiers* of the references' source and target, instead of checking whether the references actually have *the same* source and target classes according to the match model. Thus, a way is needed to access the correspondences of the respective *match model* computed in the preceding model matching step of the versioning process (cf. Figure 7) in addition to the ECL operation `matches`.

```
1  rule EReference2EReference
2    match left : Left!EReference
3    with right : Right!EReference {
4    compare :
5      left.name = right.name and
6      left.eContainer.prevMatches(right.eContainer) and // prevMatches() instead of matches()
7      left.eType.prevMatches(right.eType) // prevMatches() instead of matches()
8  }
9
10 // Substitute operation for matches() to query the match model of the
11 // preceding match instead of invoking another comparison rule.
12 @cached
13 operation Any prevMatches(opposite : Any) : Boolean {
14   var theMatch := prevMatchTrace.getMatch(self, opposite);
15   if (theMatch <> null) {
16     return theMatch.matching;
17   } else {
18     return false;
19   }
20 }
```

Listing 2 – Accessing the Match Model from within Signifier Specifications

To allow for accessing the match model of the preceding UUID-based model matching step from ECL match rules, we translate the match model of the first step of the versioning process into an ECL match trace and add it to the execution context of ECL with a variable named `prevMatchTrace`. As a result, this variable is visible to all signifier specifications and can be used to query the correspondences of model elements according to the preceding match model. To make the signifier specification more convenient, we further add an ECL operation called `prevMatches` to the execution context, which can be used instead of the operation `matches`, to examine whether two model elements correspond to each other. The accordingly adapted signifier specification for `EReference`, as well as the added ECL operation `prevMatches` is given in Listing 2. As in this adapted signifier specification the operation `prevMatches` is employed instead of `matches` (cf. line 6 and 7 in Listing 2), the match model of the preceding model matching step is now used instead of other match rules to examine whether the references' source and the target are the same (and not only share the same signifier). Thereby, we allow to isolate the comparison of signifiers from other signifier specifications. Nevertheless, as the operation `matches` is still available, users may still de-couple the comparison logics across several match rules if needed.

5.3 Signifier Preprocessing

For detecting merge issues based on signifier specifications, we have to compare the signifiers of model elements in all three considered versions of a model (V_o, V_{r1}, and V_{r2}) with each other. The computation of signifiers, however, might be rather time-consuming. To avoid having to compare *all* model elements with each other, we aim at keeping the number of signifier comparisons at a minimum. In fact, we only have to compare the signifiers of model elements if they have been added in the course of a

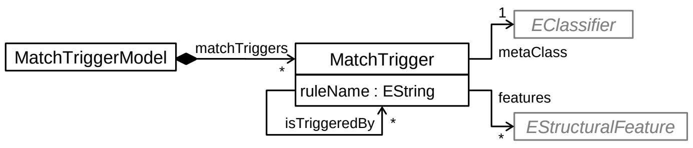


Figure 8 – Match Trigger Metamodel

revision or if they have been subjected to modification such that their signifiers might have changed. In the context of the versioning process, the information on the applied changes is available in terms of difference models, which we may exploit to prune the necessary signifier comparisons. Thus, we apply a preprocessing step, which analyzes the signifier specifications to reveal and represent explicitly the metamodel features of a model element type that constitute the respective signifier. Being equipped with this information, we only have to compare those model elements that have been changed at those features.

For collecting the information on the metamodel features that are read by the ECL rules realizing the signifiers, we traverse through the abstract syntax tree (AST) of each rule. If we reach a node in this tree that corresponds to a feature name of the modeling language's metamodel, it is saved as a visited feature for the metamodel type that is compared by the current rule. If we encounter the built-in operation `matches`, we additionally have to keep track of the metamodel features that are visited by the rule that is indirectly invoked by this operation call. Therefore, we read the argument that is passed to the `matches` operation and obtain the argument's metamodel type. Then, we search for the corresponding rule that processes the obtained metamodel type and also save the features that are visited by the obtained rule. Additionally, we have to regard invocations of user-defined ECL operations and obtain the features of model elements that are read in the called operations.

The obtained visited features are saved in terms of another model. Therefore, we introduce the *match trigger metamodel* depicted in Figure 8. This metamodel contains a root class called MatchTriggerModel, which contains for each rule in the signifier specification a dedicated instance of the class MatchTrigger. Match triggers refer to the EClassifier from the modeling language's metamodel that is processed by the current rule and the features that are visited by the current rule for computing the signifier of the respective EClassifier. As rules may depend on each other in terms of calls to the operation matches, a match trigger may also be triggered by other match triggers. For instance, if in the ECL rule for Ecore references the operation matches is called for the containing Ecore class, the match trigger for the Ecore references refers to the match trigger for Ecore classes through the reference isTriggeredBy. Thereby, we know that we have to run the signifier comparison for an Ecore reference either if its name or the respective features of the containing class have been changed. Please note that we only represent a conservative estimation of the actual impact an applied change has on a model element's signifier, as we do not consider the full semantics (such as logical conjunctions) of ECL rules.

The usage of match triggers is further illustrated in terms of a small example, which is depicted in Figure 9. In this figure, we show the match trigger model for the ECL rule EReference2EReference from the signifier specification in Listing 2. This ECL rule considers the three features, name, eContainer, and eType, for computing the signifier of instances of EReference. Accordingly, the corresponding match trigger model contains one instance of MatchTrigger, which refers to the metamodel class

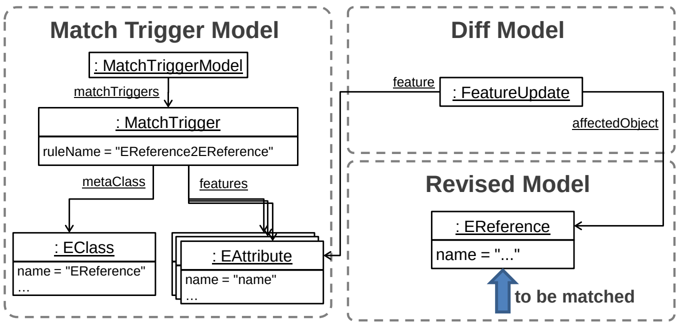


Figure 9 – Example for Match Trigger

EReference and the three features of EReference that are relevant for the signifier computation. Next to the match trigger model, Figure 9 also shows an excerpt of a difference model (e.g., $D_{V_o,V_{r1}}$), as well as an excerpt of a revised model (e.g., V_{r1}). The difference model contains one change in terms of an instance of FeatureUpdate. This instance describes the applied feature update by referring to the modified object in the revised model (cf. reference affectedObject) and the modified feature (cf. reference feature). As the instance of FeatureUpdate denotes a modification of the feature name which is also referenced from the match trigger, we may determine that the signifier of the modified object in the revised model might have been affected by the feature update and, thus, should trigger the matching process for the modified object.

5.4 Contradicting Change of a Model Element's Meaning

Contradicting changes of the meaning of model elements denote scenarios, in which the signifier of a model element is contradictorily affected in the course of both concurrent revisions (cf. Figure 6a). As a result, it is most likely that the model element's meaning is obfuscated when merging the concurrent changes naively. Therefore, the developers should be notified in order to review the concurrent contradicting change of the signifier. An example for such a scenario is discussed in Section 3 with the concurrent change of the reference createdReports with the UUID `r1`.

To detect such merge issues, we have to compare the signifiers of the model elements in the revised models, V_{r1} and V_{r2}, with the signifiers of their corresponding model elements in the origin model V_o and check whether the signifier has been affected concurrently in both revisions in a contradictory way.

Detecting contradicting signifier changes. In a first step, we select the revision having the fewest differences according to the respective difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, as we may assume that a revision with less differences also comprises less signifier changes. Having identified the smaller revision—let us assume it is V_{r1} in the following—, we iterate through all model elements in V_{r1} that have been modified in a way that their signifier might have changed as indicated by the respective match trigger (cf. Section 5.3). Next, we obtain the corresponding original model element for the revised element using the match model $M_{V_o,V_{r1}}$ and apply the signifier specifications to this pair of model elements in order to examine whether the signifier of the model element has been changed between V_o and V_{r1} (i.e., the ECL rule realizing the signifier specification does not indicate a match). For the evaluation of the signifier specifications, we add the respective match model (in this case $M_{V_o,V_{r1}}$) to the `prevMatchTrace` of the ECL execution context as discussed in Section 5.2.

If the signifier of the model element has been changed between V_o and V_{r1}, we check whether the signifier of the same original model element has also been affected in the opposite revision. Therefore, we select the corresponding revised model element of the opposite revision V_{r2} based on the match model $M_{V_o,V_{r2}}$ and compare its signifier with the signifier of the original model element in V_o. Of course, we now integrate the match model $M_{V_o,V_{r2}}$ in the variable `prevMatchTrace` for this signifier evaluation. If these two model elements do not match according to the signifier specification, we encounter a *concurrent signifier change*: the signifier of the original model element is different from the signifiers of both revised model elements.

In order to check whether the signifier has been changed not only concurrently, but also *contradictorily*, we compare the signifiers of the respective revised model elements from V_{r1} and V_{r2} *with each other*. Therefore, we first derive a new match model, called $M_{V_{r1},V_{r2}}$, from the two match models, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$, which explicates the correspondences between the revised models, V_{r1} and V_{r2}, directly. This derivation is straightforward, as we may add a correspondence between the model element $e_{r1} \in V_{r1}$ and the model element $e_{r2} \in V_{r2}$ to $M_{V_{r1},V_{r2}}$, if and only if an original model element e_0 corresponds to e_{r1} according to $M_{V_o,V_{r1}}$ and the same original model element e_0 corresponds to e_{r2} according to $M_{V_o,V_{r2}}$. This computed $M_{V_{r1},V_{r2}}$ is integrated in the ECL execution context using the variable `prevMatchTrace` for this evaluation. Next, we evaluate whether the signifier specifications of the respective model element in V_{r1} matches with the signifier of the corresponding model element in V_{r2}. If their signifier are again different, a *contradicting* signifier change is added to the conflict model C_{m_1,m_2}, which is handed over to the resolution & merge step (cf. Figure 7).

Resolving contradicting signifier changes. Contradicting changes of a model element's signifier are not severe conflicts in the common sense, as they do not denote directly interfering operations that cannot be merged. If the preceding conflict detection step in the versioning process does not report otherwise, a unique merged model can still be created, despite the contradicting change of a model element's signifier. Thus, such merge issues can be thought of as *merge warnings* with the goal of pointing the developers' attention to a *potentially obfuscated meaning.*

If developers are confronted with such a warning, the model versioning system can offer three resolution strategies. The first one is to simply drop the warning and merge. Thereby, the developers indicate that the contradicting signifier change is a "false alarm" and the merge should just proceed as normal. Second, as with usual merge conflicts, the developers may decide to omit one of the contradicting changes. That is, either the version of the contradictorily affected model element of V_{r1} or V_{r2} is used in the merged model and the other version of the model element is dropped. Third, the developers may decide that both meanings of the model element are actually contradicting and both meanings should be reflected in the merged model. Thus, the model element is split so that one model element represents the meaning of V_{r1} and the other one reflects the meaning of V_{r2}. We proposed to apply this resolution strategy for the references createdReports and managedDevelopers in our example presented in Section 3.

5.5 Overlapping Meaning of Different Model Elements

An overlapping meaning of model elements occurs if the signifier of a model element in one revised model matches unexpectedly the signifier of another model element in the opposite revised model (cf. Figure 6b). By *unexpected*, we mean that the two model elements have the same signifiers, although they actually have *no common origin model element* according to the match model (i.e., they do not correspond to each other with respect to UUIDs). As a result, when naively merging the operations of both revisions, we end up having two model elements that represent the same referent, thus the merged model seems to have redundant model elements. As an example for such a scenario, we refer to the concurrent insertion of the reference assignee in Section 3.

Detecting unexpected signifier matches. To avoid such an unfavorable redundancy, we compare the signifier of model elements in the revised models, V_{r1} and V_{r2}, *with each other*. Therefore, we have to match the signifiers of *all inserted* model elements in V_{r1} and V_{r2}, as well as those that may have *changed* their signifier according to the match trigger (cf. Section 5.3), with each other. Please note that this information is available already from detecting contradicting signifier changes (cf. Section 5.4). Unchanged model elements do not have to be checked, as we assume that there has been no redundancy in the original model already. Of course, we only need to compare model elements of the same type with each other. Furthermore, we may omit those model elements that correspond to each other according to the match model, because they are *expected* to have the same signifier.

Thus, we iterate through the revised model V_{r1} and select, for each inserted or accordingly modified model element, all inserted or accordingly modified model elements of the opposite revised model V_{r2} that have the same type. Next, we apply the signifier specifications to check whether the respective model element from V_{r1} has the same signifier as one of the previously selected opposite model elements in V_{r2}. For evaluating the signifier specifications, we again integrate the derived match model $M_{V_{r1},V_{r2}}$ (i.e., the derived direct correspondences between V_{r1} and V_{r2}) using the variable `prevMatchTrace`. If the signifiers of two model elements match, we encounter an *unexpected matching signifier*.

It is helpful for the resolution & merge steps in the versioning process to know whether the matching model elements are entirely equal (i.e., all their feature values are the same) or whether they are "only" similar. By similar, we mean that their signifier matches, but at least one of their feature values is different. Thus, in the final step, we perform a fine-grained comparison of the matching model elements to examine whether they are equal or similar and add a description of the unexpected signifier match to the conflict model C_{m_1,m_2}.

Resolving unexpected matching signifiers. Unexpected matching signifiers are not severe conflicts, because the concurrent changes are not interfering directly. However, they are a strong indicator of an unfavorable redundancy if the model is merged naively. Thus, a merge issue in the form of a warning is raised to indicate the potential redundancy.

A model versioning system may offer two different resolution strategies to resolve such a merge issue. As this issue is not a severe conflict, but rather a merge warning, the developers may simply decide to drop the warning if these two model elements actually have different meanings. Note that it is not necessary to save this decision for future revisions, because unmodified model elements are not incorporated in the signifier comparison. Thus, we only have to remove the warning from C_{m_1,m_2}. If, on the contrary, the matching model elements actually exhibit an overlapping meaning, the developers may decide to merge the two model elements into one model element. However, if the two unexpected matching model elements are not *entirely equal* but only share a common signifier, the model versioning system has to ask the involved developers for a decision concerning each different feature value of both model elements. After these decisions are taken, merging the model elements is largely straightforward. The model versioning system has to accept either the left or the right version of the matching model elements, apply the feature values according to the developers' decisions, and drop the other model element. However, potentially existing incoming links and containments of the dropped model element have to be incorporated into the accepted model element.

6 Case-based Evaluation and Critical Discussion

In this section, we study the applicability of our approach in terms of a case-based evaluation. Therefore, we show how the motivating example of Section 3 can be solved using the detection mechanisms proposed in this paper. Finally, we conclude this section with a critical discussion of the general approach in which we compare its benefits with related work and analyze its performance.

6.1 Solving the Motivating Example

In Section 3, we presented a versioning scenario illustrating several challenges for merging concurrent revisions of models that cannot be solved by existing approaches. In this section, we demonstrate how this versioning scenario can be addressed using the signifier-based merge approach. Therefore, we step through each issue and discuss how it is detected and resolved.

As discussed in Section 5.1, we apply a UUID-based model differencing approach as a basis for the signifier-based merge issue detection. Thus, the identities of model elements are preserved across different revisions of the model. When applying such an approach to the versioning scenario presented in Section 3, the merged model exhibits three issues: (*i*) the obfuscated meaning of the reference managedDevelopers, (*ii*) the redundant reference assignee, and (*iii*) the reference createdReports, which resides in the class Manager instead of the class Developer in the merged model (cf. Figure 2a).

(*i*) **Contradicting signifier change.** In Figure 10, we extracted the relevant parts of the versioning scenario leading to the obfuscated meaning of the reference managedDevelopers. The original reference createdReports has been moved from Manager to Developer in the course of m_1 resulting in V_{r1}. With m_2 the name, as well as the target of the same reference, has been changed such that the resulting reference is named managedDevelopers going from Manager to Developer (cf. V_{r2} in Figure 10). When merging these changes naively, the meaning of the reference is obfuscated, because we obtain a reflexive reference in the class Developer.

When applying the proposed mechanism for detecting contradicting signifier changes based on the signifier specification in Listing 2, the reference createdReports of V_o is compared to the corresponding reference in V_{r1}. As the source (i.e., the feature `eContainer`) of this reference has been changed in this revision, a signifier change

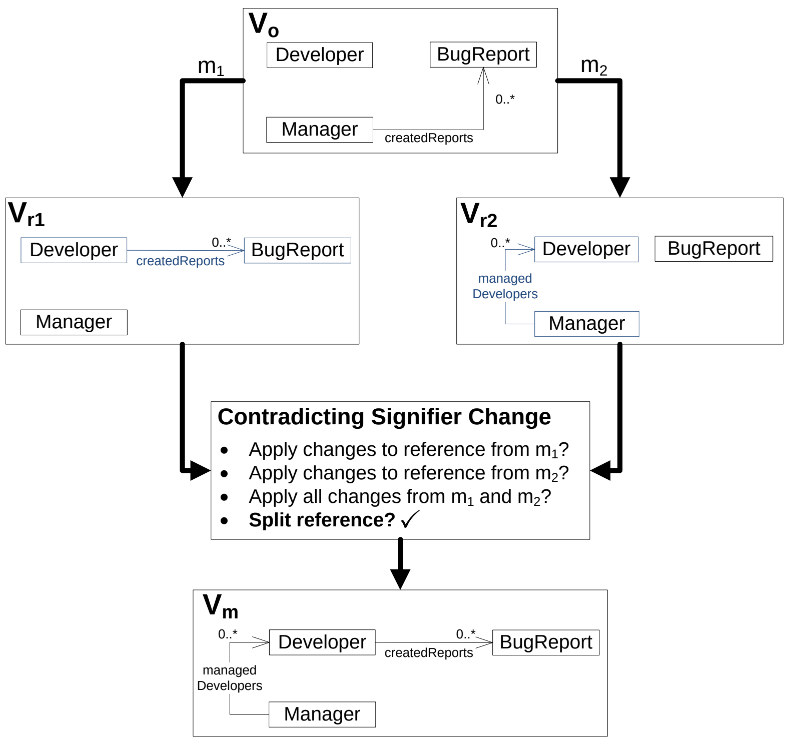


Figure 10 – Resolving the Contradicting Signifier Change

is detected because line 6 of the signifier specification in Listing 2 returns false, as the class Manager does not correspond to the class Developer according to the match model. Thus, in the next step, the same reference in V_o is matched with its corresponding reference in V_{r2}. In this revision, the reference's name, as well as its target has been modified, which also causes a signifier change between V_o and V_{r2}. Thus, we compare the signifiers of the reference in V_{r1} with the signifier of the reference in V_{r2}. As neither the reference's name, nor its source, and its target correspond to each other, a *contradicting signifier change* is reported.

To resolve this merge issue, the model versioning system offers four resolution strategies, which are depicted in the center of Figure 10. The corresponding references in V_{r1} and V_{r2} have in fact divergent meanings and both meanings should be reflected in the merged model. Hence, the developer decides to split the references, which avoids obfuscating the reference's meaning and leads to a superior merged model V_m as depicted in Figure 10.

(ii) **Unexpected signifier match at reference assignee.** For illustrating the detection of the second issue, namely the concurrent insertion of the reference assignee, we depict the relevant model elements of the versioning scenario in Figure 11. The original version contains two classes, namely Developer and BugReport. Both developers add a reference called assignee to the class BugReport, which refer to the class Developer.

The detection mechanism for unexpected signifier matches compares all model elements of V_{r1} and V_{r2} to each other that have been added or potentially changed

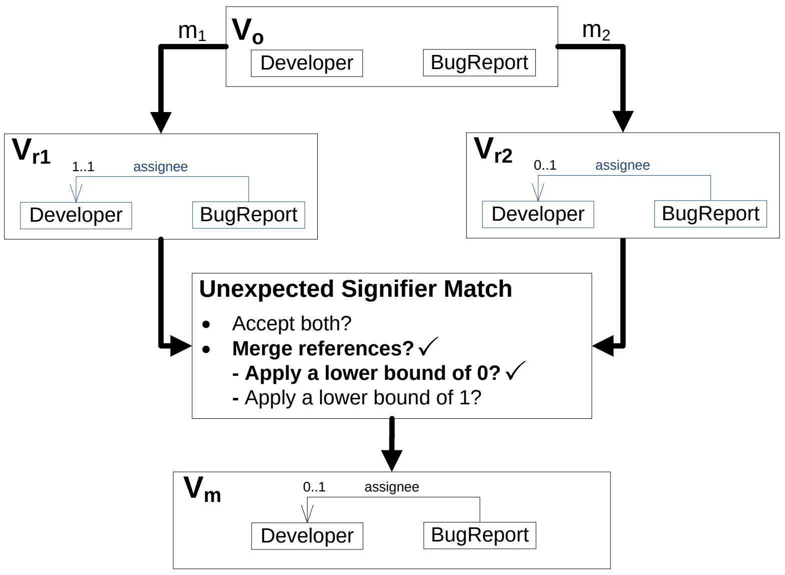


Figure 11 – Resolving the Unexpected Signifier Match of assignee

their signifier. Hence, the ECL rule in Listing 2 is applied to the reference assignee of V_{r1} and of V_{r2}. As their source and target classes correspond to each other and their name is equal, an unexpected signifier match is reported. The model versioning system may now offer two resolution strategies: the warning is either ignored, i.e., both are introduced to the merged version, or the model elements are merged. We may assume that the developer decides to merge the two references, because both reflect the same entity. However, the references to be merged are not entirely equal; thus, the model versioning system asks for a decision concerning each different feature value, which is, in our scenario, only the lower bound of the reference. Supposing the developer decides to accept a lower bound of `0`, we obtain V_m in Figure 11.

(*iii*) **Redundant reference createdReports.** Besides the signifier-based merge warnings mentioned above, there is another issue concerning the redundant references named createdReports. As depicted in Figure 12, developer 1 specified Manager to be a subclass of Developer and moved the reference createdReports from Manager to Developer. In the concurrent revision, however, developer 2 created a copy of the same reference, but left the copy (having the UUID r3) unchanged in the class Manager, but renamed and moved the original reference `r1` to realize the reference managed-Developers in Manager. Consequently, developer 2 implicitly added the new reference createdReports with the UUID `r3` to the class Manager.

When merging all changes and resolving the contradicting signifier change of the reference having the UUID `r1` by splitting it as discussed before, we end up with a merged model that contains the reference createdReport `r1` in the class Developer, as well as the equally named reference `r3` in the class Manager. As the reference with the UUID `r1` has been subject to change at its feature `eContainer` and the reference with the UUID `r3` has been added in the concurrent revision, the signifiers of these two references are compared in the detection process for unexpected signifier changes. However, the signifiers do not match because their containers (i.e., `eContainer`) do not correspond to each other. Nevertheless, these references are redundant, because the class Manager *inherits* the reference createdReports and contains an equally named reference itself. Unfortunately, our current signifier specification does not consider inheritance among Ecore classes.

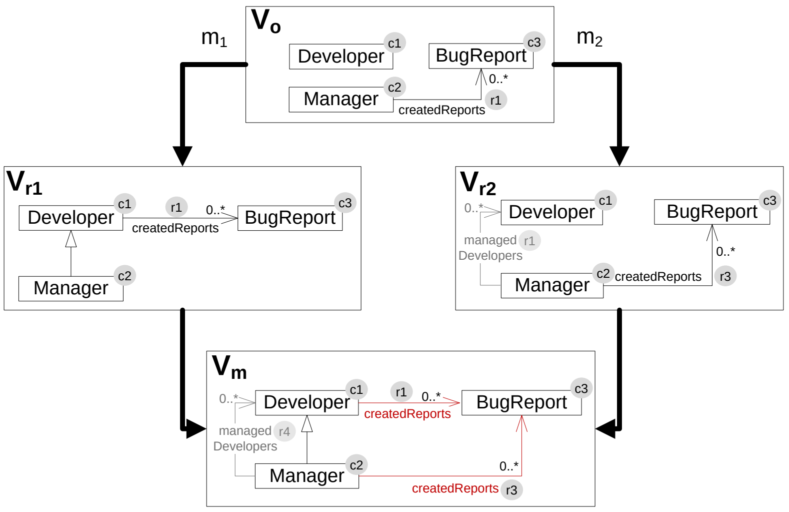


Figure 12 – Redundant Reference createdReports

```
rule EReference2EReference
  match l : Left!EReference
  with r : Right!EReference {
  compare :
    l.name = r.name and
    l.eContainer.prevMatchesOrSuperType(r.eContainer) and
    l.eType.prevMatchesOrSuperType(r.eType)
}

operation EClass prevMatchesOrSuperType(other : EClass) : Boolean {
  if (self.prevMatches(other)) { return true; }
  for (superType in prevMatchTrace.getMatch(self).getRight().eAllSuperTypes) {
    if (superType = other) {
      return true;
    }
  }
  for (leftSuperType in self.eSuperTypes) {
    if (leftSuperType.prevMatchesOrSuperType(other)) {
      return true;
    }
  }

  for (superType in prevMatchTrace.getMatch(other).getLeft().eAllSuperTypes) {
    if (superType = self) {
      return true;
    }
  }
  for (rightSuperType in other.eSuperTypes) {
    if (self.prevMatchesOrSuperType(rightSuperType)) {
      return true;
    }
  }
  return false;
}
```

Listing 3 – Respecting Inheritance in Signifier Specifications

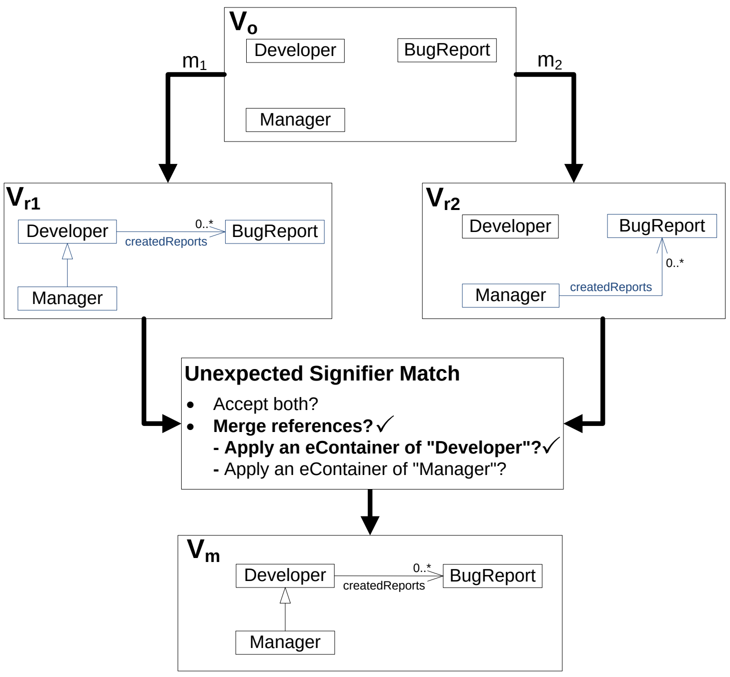


Figure 13 – Resolving the Unexpected Signifier Match Regarding Inheritance

Incorporating inheritance of Ecore classes. To integrate the concept of inheritance of Ecore classes in the signifier specification for Ecore references, we have to extend the specification as depicted in Listing 3. In particular, we added the operation `prevMatchesOrSuperType` and use it instead of the operation `prevMatches` for comparing the `eContainer` and `eType` references. This new operation checks, on the one hand, whether the specified instances of `EClass` correspond to each other directly according to the injected match model and, on the other hand, whether the one class is specified to be a superclass of the other in one of the compared models or vice versa. Therefore, we first obtain the corresponding class of `self` (i.e., the class of the left model) from the right model in line 12 using the injected match model and iterate through all its superclasses. If one of the superclasses is equal to `other` (i.e., the class in the right model), we determined that `other` is specified to be a superclass of `self` in the right model and return `true`. Subsequently, we check whether `other` is the superclass of one of the superclasses of `self` recursively (cf. line 18). Next, we evaluate the same in the left model (cf. line 23–32), because the inheritance relationship might have been specified either in the left or the right revision. Thereby, `a.prevMatchesOrSuperType(b)` returns true, if `b` is a direct or indirect superclass of `a` either in the left or right model, or if they correspond to each other directly.

The benefits of this extended signifier specification is twofold. First, no signifier change is reported if a developer pulls a reference to a superclass, which is an improvement because the meaning has not been affected by this refactoring. Second, we also obtain an unexpected signifier match, if the source classes of references are two different classes, whereas one class inherits the reference from the other. Consequently, we may also solve the last issue of our versioning scenario concerning the indirectly redundant reference createdReports as depicted in Figure 13: the reference

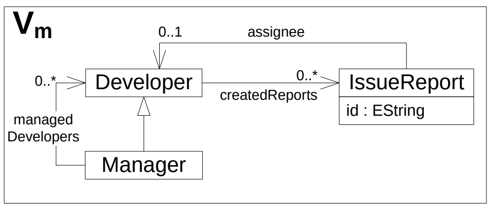


Figure 14 – Final Result of the Resolution

createdReports in V_{r1} now exhibits the same signifier as the equally named reference in V_{r2}, because in V_{r1}, the source class of the reference is a superclass of Manager and, in V_{r2}, the class Manager acts as source of the reference itself. Thus, a merge issue is raised by the model versioning system to indicate an unexpected signifier match. As the matching references have different feature values at the feature `eContainer`, the developer may now decide to use the class Developer as `eContainer` of the merged reference. Ultimately, after resolving all raised issues as discussed in this section, the obtained merged version corresponds to the optimal merged version depicted in Figure 14, which is of much better quality than the merge that is obtained from existing model versioning systems (cf. Figure 2).

6.2 Critical Discussion

By solving successfully the issues posed by the versioning scenario in the previous section, we believe that the presented approach for extending generic model versioning systems has the potential to address several challenging merge issues in a novel way. Considering that ECL provides several more powerful possibilities, such as integrating external Java libraries, we may even realize fuzzy signifiers based on Levenshtein distance [Lev66] or thesauri, which might be useful in some scenarios.

Critical comparison to related work. The versioning systems ADAMS [DLFST09] and Odyssey-VCS [MCPW08] allow to specify a so-called *unit of comparison*, which can be thought of as the granularity of the conflict detection. More precisely, when the unit of comparison is set to the model element level instead of feature level, the versioning system will report a conflict for any concurrent changes to the same model element, irrespective of whether or not the concurrent changes modified the same features of the model element. In our example, this would lead to a conflict at the reference with the UUID `r1`, because one developer changes its name and its target and the other developer moves the same reference into another class. Although this is the desired result in this scenario, we argue that this is too coarse-grained in general, as *every* kind of concurrent change of the reference results in a conflict and not only those changes that affect the reference's meaning in a contradicting way. For instance, if one developer changes the reference to `unordered` and the other one sets it to be `transient`[4], this would also cause a conflict when setting the unit of comparison to the entire `EReference`. However, such changes could be merged easily without obtaining an obfuscated meaning of the reference.

A much more flexible approach, in comparison to ADAMS and Odyssey-VCS, has been proposed by Cicchetti et al. [CDRP08]. The approach by Cicchetti et al.

[4]`ordered` and `transient` are linguistic attributes [KKK+06] of references and attributes in Ecore.

allows to specify conflict patterns, which indicate any combination of changes that shall be reported as conflict. Their approach is more general and addresses a broader set of conflicts, whereas our approach is targeted at a specific set of language-specific conflicts. Nevertheless, their approach may allow to specify and detect at least a subset of the merge issues discussed in this paper. The underlying concepts of the approach by Cicchetti et al. are very different from ours. Whereas Cicchetti et al. follow a change-based approach for specifying conflicts in terms of forbidden pairs of changes, we use declarative signifier specifications, which are independent of the actual changes that cause the respective merge issues. Thus, signifiers provide a more concise and domain-specific way that abstracts from applied changes. In fact, from a signifier specification it would be possible to infer all relevant conflict patterns (i.e., the crossproduct of all changes that may affect the signifier concurrently). Unexpected overlapping meanings of model elements have not been considered by Cicchetti et al.

Performance. The presented signifier-based approach has probably a negative affect on the overall performance, because it requires several potentially time-consuming comparisons. To assess the performance of our approach, we prototypically realized the signifier-based merge issue detection and integrated it with the model versioning system AMOR. The prototypical implementation is available at EclipseLabs[5]. The automatic derivation of match triggers from ECL rules presented in Section 5.3 is, however, still work in progress; currently, they have to be encoded manually.

For evaluating the impact of the signifier-based merge issue detection on the execution time of AMOR, we simulated several model versioning scenarios by applying 30 changes in each revision randomly to five different original models. Please note that 30 concurrent changes is significantly higher than the average number of changes per commit according to the case study performed by Herrmannsdoerfer et al. [HRW09]. With these versioning scenarios, we measured the steady state performance[6] of AMOR including and excluding the signifier-based merge issue detection. The five different original models are Ecore metamodels containing from 992 model elements up to 84.570 model elements for covering a wide range of differently sized models. The signifier specification contained four ECL rules; i.e., for `EPackage`, `EClass`, `EAttribute`, and `EReference`, whereas the name, the container, and in case of `EAttribute` and `EReference` also the type has been incorporated in the signifier specification. For matching the container and type, we used the operation `prevMatchesOrSuperType` given in Listing 3. Note that we exploited the caching mechanism of ECL for all rules and operations (i.e., `@cached` [KRP11]).

The results of the performance evaluation of our prototype are plotted in Figure 15. This figure shows the execution time in milliseconds (ms) of AMOR excluding the signifier-based merge issue detection (denoted with Conflict Detection in the legend of Figure 15), as well as the additional execution time required for detecting contradicting signifier changes and unexpected signifier matches. Furthermore, the overall sum of the execution time of AMOR including both signifier-based detection mechanisms is represented. AMOR uses a model differencing algorithm based on synthetic identifiers. Although this is much more efficient than heuristic approaches, its execution time still depends heavily on the size of the compared models. Also, the additional execution time required for the signifier-based merge issue detection grows with the number of model elements. However, this is not because more model elements have

[5]`http://code.google.com/a/eclipselabs.org/p/amor-conflict-detection`

[6]A program is run repeatedly until the execution time of each run stabilizes.

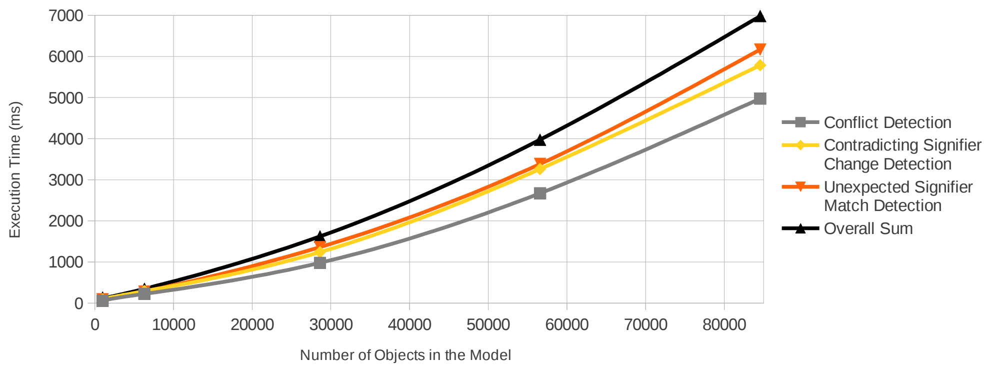


Figure 15 – Results of the Performance Evaluation

to be compared, since only inserted and correspondingly modified model elements are included in the signifier matching. Nevertheless, the more model elements are in the model, the larger is the computed match model. Thus, deriving the match traces that are incorporated in the ECL execution context consumes more time and the invocations of the operations `prevMatches` and `prevMatchesOrSuperType` become more expensive with an increasing number of model elements.

In summary, we argue that the additionally required execution time is reasonable in comparison to the benefits that are accomplished by the signifier-based merge issue detection. For instance, when applying the presented detection mechanisms to a model of over 6.000 model elements, the additional execution time is only 123 ms with 30 changes in each revision. The additional execution time for a model containing more than 28.000 model elements constitutes 642 ms. However, a relatively large model with more than 84.000 model elements caused the additional execution time to increase up to 2 seconds. In our opinion, this is still acceptable when considering that a model differencing approach that is based on heuristics only, has a much worse performance. For instance, when applying the heuristic-based matching algorithm of EMF Compare [BP08] to the model containing around 28.000 model elements, it takes already more than 2 minutes to compute the differences.

Threats to validity. In this section, we evaluated the applicability of the proposed approach by investigating a model versioning scenario containing several challenging merge issues and assessed the performance of our prototypical implementation. Although both experiments showed very promising results, there are some threats to validity, which are discussed in the following.

As every case-based evaluation, the validity of the experiment presented in Section 6.1 might suffer from a lack of comprehensiveness. Even though we gained promising results also from investigating several scenarios other than the one presented in this paper (cf. [Lan11]), a more extensive evaluation based on real-world scenarios would allow for deeper insights in the benefits of the presented approach. However, such an extensive evaluation is, due to the lack of empirical data, not easy to perform and has been left for future work.

Furthermore, the validity of the performance evaluation might be affected, as we used synthetic models and not real-world models. To mitigate this threat, we generated models having a comparable fraction of numbers of packages, classes, attributes, references, and inheritance relationships as the UML 2 metamodel. More precisely, in the generated models, around 15 % of all model elements are classes, 60 % are references, and 25 % are attributes. At least seven out of ten classes have at least one superclass. The maximum inheritance depth of the five generated models are 4, 22, 29, 77, and 121.

Realization alternatives for signifiers. The implementation of the approach is realized based on EMF and ECL. However, the conceptual approach is portable to other technological spaces, metamodeling frameworks, and languages. Therefore, one would only have to realize the means for computing and comparing feature values constituting a model elements' signifier. To avoid a poor performance, a technique comparable to match triggers should be considered in addition. For instance, one could use the Object Constraint Language (OCL) [OMG10], a specification language for UML standardized by the Object Management Group (OMG), instead of ECL to specify signifiers. This would also allow for exploiting existing OCL tools and algorithms, such as the incremental evaluation of OCL constraints [BHR$^+$10, CT09] to maintain a reasonable performance. Nevertheless, when using plain OCL for specifying and comparing signifiers, one would have to implement an infrastructure for matching model elements based on OCL statements, a match tracing mechanism, etc. Although this would be possible, we preferred reusing the already available infrastructure provided by ECL for implementing our prototype.

7 Conclusion and Future Work

In this paper, we identified several merge issues that are insufficiently supported by current generic model versioning systems. To address these issues, we proposed an *orthogonal extension* of *generic model versioning systems*. Therefore, we introduced the notion of *signifiers* representing the combination of properties of model elements that convey its superior meaning and showed how such signifiers can be realized and integrated in the versioning process. We validated the applicability of this approach by showcasing its benefits for merging Ecore models. We are aware that the presented approach does not enable the detection of *all* possible language-specific merge issues. However, we believe that signifiers facilitate a *lightweight approach* to specify important aspects of modeling languages and that integrating them into a versioning system enables the detection of an *important subset* of *language-specific merge issues*.

There are several lines for future work. First, we plan to develop a semi-automatic process for deriving signifier specifications from metamodels and models. Therefore, we may leverage several sources of information in the context of modeling languages, such as uniqueness constraints in the metamodel, validation rules, as well as the concrete syntax specification, as an indicator for important features of a model element type. Moreover, we aim for a more convenient specification of signifiers directly on top of the metamodel's concrete syntax using a model annotation mechanism, such as EMF Profiles [LWWC12]. Furthermore, we also plan to investigate techniques to attach user-specified resolution strategies to signifier specifications (e.g., using the Epsilon Merging Language [EPK06]). Another line of future research is the evaluation of an signifier enhanced version of a model comparison framework with respect to precision and recall using a benchmark as proposed by van den Brand et al. [vdBHVP11]. Finally, we plan to investigate the usefulness of signifiers in other fields, e.g., to detect equivalence correspondences between models in two-way merge scenarios [CNS12] such as model composition and weaving or to reason about the identities of model elements in bi-directional transformations [HLR08].

References

[AK03] C. Atkinson and T. Kühne. Model-driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, 2003. doi:10.1109/MS.2003.1231149.

[AP03] M. Alanen and I. Porres. Difference and Union of Models. In *Proceedings of the 6th International Conference on the Unified Modeling Language (UML'03)*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003. doi:10.1007/978-3-540-45221-8_2.

[BHR$^+$10] G. Bergmann, A. Horvath, I. Rath, D. Varro, A. Balogh, Z. Balogh, and A. Ökrös. Incremental Evaluation of Model Queries over EMF Models. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 76–90. Springer, 2010. doi:10.1007/978-3-642-16145-2_6.

[BKL$^+$12] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. An Introduction to Model Versioning. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*, pages 336–398. Springer, 2012. doi:10.1007/978-3-642-31753-8_14.

[BKS$^+$10] P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, and P. Langer. Adaptable Model Versioning in Action. In *Tagungsband der Modellierung 2010*, volume 161 of *LNI*, pages 221–236. GI, 2010.

[BP08] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008. doi:10.1016/j.ejso.2009.08.008.

[CDRP08] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008. doi:10.1007/978-3-540-87875-9_23.

[CNS12] M. Chechik, S. Nejati, and M. Sabetzadeh. A Relationship-based Approach to Model Integration. *Innovations in Systems and Software Engineering*, 8(1):3–18, 2012. doi:10.1007/s11334-011-0155-2.

[CT09] J. Cabot and E. Teniente. Incremental Integrity Checking of UML/OCL Conceptual Schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009. doi:10.1016/j.jss.2009.03.009.

[DLFST09] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora. Concurrent Fine-Grained Versioning of UML Models. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 89–98. IEEE, 2009. doi:10.1109/CSMR.2009.35.

[DS16] F. De Saussure. Nature of the Linguistic Sign. *Course In General Linguistics*, 1916.

[EPK06] K. Engel, R. Paige, and D. Kolovos. Using a Model Merging Language for Reconciling Model Versions. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and*

Applications (ECMDA-FA'06), volume 4066 of *LNCS*, pages 143–157. Springer, 2006. `doi:10.1007/11787044_12`.

[FR07] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of the Workshop on the Future of Software Engineering @ ICSE'07*, pages 37–54. IEEE Computer Society, 2007. `doi:10.1145/1253532.1254709`.

[GJM02] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[GKLE10] C. Gerth, J. M. Küster, M. Luckey, and G. Engels. Precise Detection of Conflicting Change Operations Using Process Model Terms. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 93–107. Springer, 2010. `doi:10.1007/978-3-642-16129-2_8`.

[HK10] M. Herrmannsdoerfer and M. Koegel. Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 76–81. ACM, 2010. `doi:10.1145/1826147.1826161`.

[HLR08] T. Hettel, M. Lawley, and K. Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *LNCS*, pages 31–45. Springer, 2008. `doi:10.1007/978-3-540-69927-9_3`.

[HRW09] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language Evolution in Practice: The History of GMF. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09)*, volume 5969 of *LNCS*. Springer, 2009. `doi:10.1007/978-3-642-12107-4_3`.

[KHWH10] M. Koegel, M. Herrmannsdoerfer, O. Wesendonk, and J. Helming. Operation-based Conflict Detection on Models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 21–30. ACM, 2010. `doi:10.1145/1826147.1826154`.

[KKK$^+$06] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting Metamodels to Ontologies–A Step to the Semantic Integration of Modeling Languages. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 528–542. Springer, 2006. `doi:10.1007/11880240_37`.

[Kol09] D. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proceedings of the 5th International Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009. `doi:10.1007/978-3-642-02674-4_11`.

[KRP11] D. Kolovos, L. Rose, and R. Paige. *The Epsilon Book.* Online: `http://www.eclipse.org/gmt/epsilon/doc/book/`, 2011.

[Küh06] T. Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006. `doi:10.1007/s10270-006-0017-9`.

[Lan11] P. Langer. *Adaptable Model Versioning based on Model Transformation By Demonstration.* PhD thesis, Vienna University of Technology, 2011.

[Lev66] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LGJ07] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-specific Models. *European Journal of Information Systems*, 16(4):349–361, 2007. `doi:10.1057/palgrave.ejis.3000685`.

[LvO92] E. Lippe and N. van Oosterom. Operation-Based Merging. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environment (SDE'92)*, pages 78–87. ACM, 1992. `doi:10.1145/142868.143753`.

[LWWC12] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(4):1–29, 2012. `doi:10.5381/jot.2012.11.1.a8`.

[MCPW08] L. Murta, C. Corrêa, J.G. Prudêncio, and C. Werner. Towards Odyssey-VCS 2: Improvements Over a UML-based Version Control System. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models @ ICSE'08*, pages 25–30. ACM, 2008. `doi:10.1145/1370152.1370159`.

[MGH05] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 204–213. ACM, 2005. `doi:10.1145/1101908.1101940`.

[NMBT05] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 215–224. ACM, 2005. `doi:10.1145/1062455.1062504`.

[OMG07] Object Management Group OMG. XML Metadata Interchange 2.1.1 (XMI). `http://www.omg.org/spec/XMI/2.1.1`, 2007.

[OMG10] Object Management Group OMG. Object Constraint Language (OCL), Version 2.2. `http://www.omg.org/spec/OCL/2.2`, 2010.

[OMW05] H. Oliveira, L. Murta, and C. Werner. Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In *Proceedings of the 12th International Workshop on Software Configuration Management @ ESEC/FSE'05*, pages 1–16. ACM, 2005. `doi:10.1145/1109128.1109129`.

[OR23] C.K. Ogden and I.A. Richards. *The Meaning of Meaning.* Harcourt, Brace, 1923.

[OS05] T. Oda and M. Saeki. Generative Technique of Version Control Systems for Software Diagrams. In *Proceedings of the 21th International Conference on Software Maintenance (ICSM'05)*, pages 515–524. IEEE, 2005. doi:10.1109/ICSM.2005.49.

[RV08] J.E. Rivera and A. Vallecillo. Representing and Operating With Model Differences. In *Proceedings of the 46th International Conference on Objects, Components, Models and Patterns (TOOLS'08)*, volume 11 of *LNBIP*, pages 141–160. Springer, 2008. doi:10.1007/978-3-540-69824-1_9.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2008.

[SG08] M. Schmidt and T. Gloetzner. Constructing Difference Tools for Models Using the SiDiff Framework. In *Companion of the 30th International Conference on Software Engineering (ICSE'08)*, pages 947–948. ACM, 2008. doi:10.1145/1370175.1370201.

[SZN04] C. Schneider, A. Zündorf, and J. Niere. CoObRA – A Small Step for Development Tools to Collaborative Environments. In *Proceedings of the Workshop on Directions in Software Engineering Environments @ ICSE'04*, 2004.

[TELW12] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A Fundamental Approach to Model Versioning Based on Graph Modifications. *Software and Systems Modeling*, 2012. doi:10.1007/s10270-012-0248-x.

[vdBHVP11] Mark van den Brand, Albert Hofkamp, Tom Verhoeff, and Zvezdan Protić. Assessing the quality of model-comparison tools: a method and a benchmark data set. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice @ TOOLS'11*, pages 2–11. ACM, 2011. doi:10.1145/2000410.2000412.

[Wes10] B. Westfechtel. A Formal Approach to Three-way Merging of EMF Models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 31–41. ACM, 2010. doi:10.1145/1826147.1826155.

[XS05] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM, 2005. doi:10.1145/1101908.1101919.

About the authors

Philip Langer is a postdoctoral researcher in the Business Informatics Group at the Vienna University of Technology. His research is focused on model evolution, model transformations, and model execution in the context of model-driven engineering. For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/planger or contact him at langer@big.tuwien.ac.at.

Manuel Wimmer is a postdoctoral researcher in the Business Informatics Group at the Vienna University of Technology. Currently, he is on leave as research associate at the University of Málaga. His research interests comprise Web engineering, model-driven engineering, and model management. For further information about his research activities, please visit `http://www.big.tuwien.ac.at/staff/mwimmer` or contact him at `wimmer@big.tuwien.ac.at`.

Jeff Gray is an Associate Professor in the Department of Computer Science at the University of Alabama. His research interests include model-driven engineering, aspect-orientation, and generative programming. For further information about his research activities, please visit `http://cs.ua.edu/~gray` or contact him at `gray@cs.ua.edu`.

Gerti Kappel is a full professor in the Institute for Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Her current research interests include model engineering, Web engineering, as well as process engineering. For further information about her research activities, please visit `http://www.big.tuwien.ac.at/staff/gkappel` or contact her at `gerti@big.tuwien.ac.at`.

Antonio Vallecillo is Professor of Computer Science at the University of Málaga. His research interests include Open Distributed Processing, Model-Based Engineering, Componentware, Software Quality, and the industrial use of formal methods. For further information about his research activities, please visit `http://www.lcc.uma.es/~av` or contact him at `av@lcc.uma.es`.

Acknowledgments This work has been partially funded by the Austrian Science Fund (FWF) under grant J 3159-N23, the Spanish Research Project TIN2011-23795, and the NSF CAREER award (CCF-1052616).