Models in Conflict – Detection of Semantic Conflicts in Model-based Development[⋆]

Thomas Reiter[2], Kerstin Altmanninger[1], Alexander Bergmayr[2], Wieland Schwinger[1], and Gabriele Kotsis[1]

[1] Department of Telecooperation
Johannes Kepler University Linz, Austria
kerstin.altmanninger@jku.at | wieland@schwinger.at |
gabriele.kotsis@jku.ac.at

[2] Information Systems Group (IFS)
Johannes Kepler University Linz, Austria
[bergmayr|reiter]@ifs.uni-linz.ac.at

Abstract. To make the model-driven paradigm a widespread success, appropriate tools such as version control systems (VCS) are required to adequately support a model-based development process. However, first approaches specializing on model-based versioning, do not take into account the semantics of the artefacts they operate upon. Thus, conflict detection mechanisms are based on detecting conflicting concurrent modifications on a software artefacts syntactic representation, only, without explicitly considering the semantics the artefact stands for.
As opposed to a heavyweight approach relying on formal mathematics, we follow a lightweight approach that is based on creating views of a model that explicate a certain aspect of a modeling language's semantics. Such a view is created through a model transformation from the original model edited by the developers. Using both the original model and the created view our approach relies on graph-based comparison strategies to detect conflicts due to concurrent editing to determine so-called *syntactic* and *semantic conflicts*, respectively. Consequently, by means of various example scenarios, we demonstrate how our approach is able detect conflicts that otherwise would remain undetected.

1 Introduction

The shift from code-centric to model-centric software development places models as first class entities in "Model-driven Software Development" (MDSD) processes. A major prerequisite for the wide acceptance of MDSD are proper methods and tools as available for traditional software development, such as build tools, test frameworks or "Version Control Systems" (VCS). Considering the latter, VCS are particularly essential when the development process proceeds in

⋆ This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.

parallel such that different developers concurrently modify a model, which may result in concurrent, potentially conflicting modifications.

Such conflicting modifications need to be resolved by appropriate techniques for *model comparison*, *conflict detection*, *conflict resolution* and *merging*, which in case of a heterogeneous tooling environment are required to operate on the resulting model (i.e. state-based).

Since, as already stated, models are the first class entities in model driven development, this should not rely on text- or tree-based VCS like Subversion [1], CVS [2] or CoEd [3]. Although they offer excellent version control techniques for text-based documents, the granularity of comparison is a single line. This makes such VCS not a first-hand choice for MDSD, since for effectively dealing with conflicting modifications the logical structure of models has to be taken into account.

For dealing with concurrent modifications on models and specifically for properly identifying conflicts, it is necessary not only to consider the syntactical structure of models (i.e. the syntax of the model) but also to "understand" the model's semantics. For example, concurrent modifications on a model may not result in an obvious conflict when syntactically different parts of the model (e.g., different model elements) were edited. Nevertheless, they may interfere with each other, thus yielding an actual conflict (e.g., modification of two different model elements may have a conflicting side effect), which without considering the model's semantics would remain hidden. Furthermore, certain conflicts may only occur due to the syntactical representation of a model, since sometimes more than one possibility exists to conceptually express the same state of affairs in a modeling language. This does not necessarily lead to a conflict with respect to the model's semantics (e.g., decision nodes as well as conditional nodes in UML activity diagrams are two ways to express alternative branches in a process). Especially modeling languages offering "syntactic sugar" in the sense that convenience constructs allow to express the same meaning in varying ways, can easily give rise to the above mentioned scenario.

Whereas model comparison and model merging can be facilitated by means of existing graph-based approaches. Facilitating an "understanding" of the models during conflict detection and conflict resolution is still an open issue for models. We argue that through the definition of semantics a VCS can find conflicts more precisely during conflict detection, thus avoiding falsely indicated conflicts and finding previously undiscovered ones.

Although, in the field of programming languages some approaches have been presented [4] providing some "understanding", they typically rely on formal semantics and apply to certain languages, only. Such approaches cannot immediately be reused in the realm of models, since unfortunately, a full formal specification of the semantics underlying a modeling language is very often not feasible: one hand, many modeling languages do not have a formal semantics as these are often hard and costly to define [5], and on the other hand, in the light of a growing number of domain specific languages a flexible and more light-weight approach is desirable.

Consequently, in this paper we lay out a light-weight approach for making use of a modeling language's semantics for conflict detection, which is based on interpreting the model in a view that makes explicit certain characteristics of the original language. Thus, additionally to existing graph-based VCS our approach has the ability to detect "semantic" conflicts that are specific to the modeling language that these models conform to. The benefits of the proposed light-weight approach are threefold. Firstly, it is *open* to incorporate any modeling tool. Secondly, it is *flexible* to operate on virtually any modeling language. Finally, it is *extensible* to incorporate the semantics of interest for a certain development scenario.

2 Semantic Versioning

2.1 Conceptual Overview of Semantic Conflict Detection

A model conforms to a certain metamodel that defines the abstract syntax of a modeling language, which itself does not provide any machine-interpretable semantics. Most definitions of semantics are functions that map the abstract syntax of one language onto the abstract syntax of another well understood language or a formal semantic domain.

As already stated, the definition of semantics for a modeling language is a difficult process involving the actual formalization of the semantics and the finding of an agreement between stakeholders thereon. For the scope of our work, such full-fledged semantic definitions are out of scope. Instead, we advocate a way of specifying semantics, that is machine-interpretable and flexible enough to just represent those aspects of a language's semantics that are of special interest for concurrent development.

Therefore, in our proposed approach, the semantic mapping between a modeling languages metamodel and a metamodel representing a certain view of interest, is defined through a model transformation. The output of such a transformation is another model which conforms to the metamodel representing the semantic view of interest. Compared to the definition of semantics for programming languages [6], our model transformation-based approach is similar to a translational semantics specification, which maps the constructs of one language onto constructs of another, usually simpler language such as machine-instructions. Similarly, in our case we translate into a view that filters out a certain facet of interest for our purpose of conflict detection, only. A translational approach can be considered as a special case of denotational semantics, with the valuation functions denoting constructs of the target language instead of a purely mathematical semantic domain. Similarly, the rules of a model transformation relate the elements of the metamodel (abstract syntax) to which the original model conforms to and the elements of the metamodel representing the semantic view.

Consequently to the transformation realizing a semantic mapping, conflict detection can be carried out on both models and semantic views. Throughout the rest of this paper we will refer to conflicts that are determined purely upon the comparison of two versions of a model as *syntactical conflicts*. A *semantic conflict* is a conflict that is detected between the representations of such a model's versions in a semantic view. The actual finding of conflicts in both the original model and the view functions analogous to the detection of structural conflicts in existing graph-based versioning systems [7–9], based on the detection of concurrent modifications to the same element.

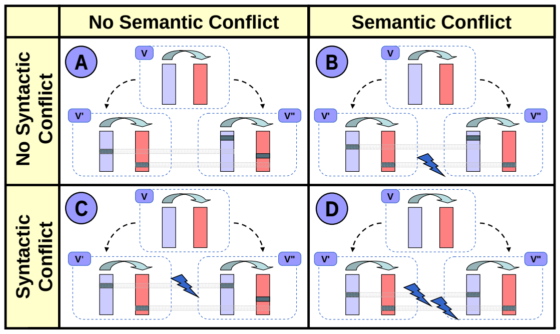


Fig. 1. Overview on conflict types

Fig. 1 shows four possible combinations of scenarios that can occur when a model's semantic views are incorporated into conflict detection. These scenarios are arranged according to an observed syntactic or semantic conflict, respectively. As a simplification, the figure does not make use of a concrete model, but uses bars as abstractions for models with highlights indicating a modification in a certain part of the model. The left bars refer to the original model (before transformation) and the right bars refer to the semantic view of a model (after transformation). Thus, if changes in the model or the view occur in the same place, a syntactic or semantic conflict is detected, respectively. Of course, only the model is checked out and edited by developers, whereas the semantic view is only a product of the transformation. The common ancestor version V is shown on top of the modified versions V' and V''.

2.2 Semantic Views by Example

In the following we introduce an example that demonstrates the definition of a semantic view for a simplified version of the "Business Process Execution Language" (BPEL) [10]. The view we are going to define focuses on certain BPEL constructs that serve as "syntactic sugar" to enhance readability and are a convenient way for structuring groups of activities. The *Sequence* construct for instance, denotes sequential execution of its contained *Activities*. However, the same meaning can be expressed by linking up individual activities accordingly. For instance, a *Sequence* S containing the *Activities* $A1$ and $A2$ is equivalent to *Activities* $A1$ and $A2$ connected by a *Link* from $A1$ to $A2$. Assuming these two semantically equivalent models originate from a common ancestor and are the outcome of concurrent editing, a conventional versioning system would find a number of differences between these two models and according to that would report a number of conflicts. A developer would then have to interpret these models and come to the conclusion that besides the structural difference between the models, no semantic difference exists. Drawing such conclusions can automatically be achieved by a comparison of models transformed into semantic views, which abstracts from syntactical modifications and allows seeing conceptual changes to the model, only.

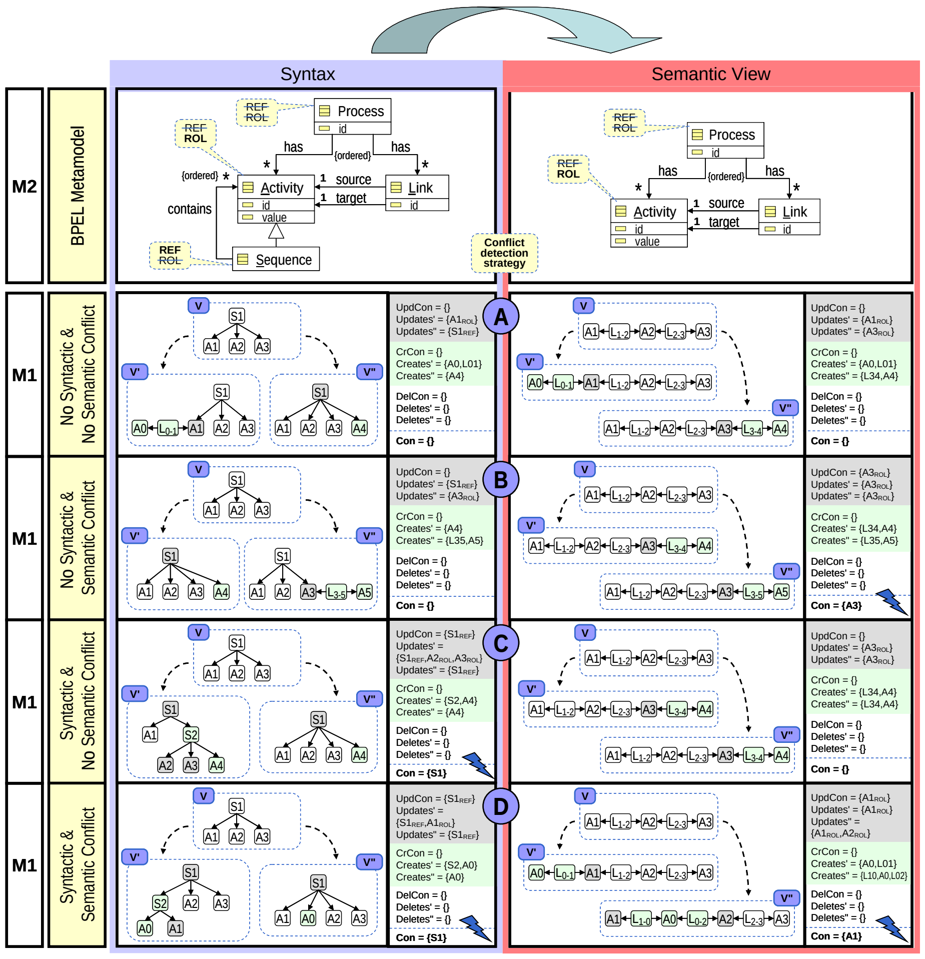
Fig. 2 shows four concrete situations where an instance of a simplified BPEL metamodel has been checked out by both developers, each time undergoing modifications that lead to one of the four conflict scenarios mentioned earlier. In every scenario the updated model elements are marked and the right hand side gives details about the calculation of elements in conflict.

In each of the scenarios, two developers make concurrent modifications to a BPEL *Sequence* $S1$ initially containing *Activities* $A1$, $A2$, and $A3$, possibly resulting in update-update conflicts. The actual comparison of model elements is based on an identifying attribute designated in the metamodel. By inspecting the structural features, namely the attributes and references of a model element, one can determine whether the model element as a whole has been updated. In particular, we differentiate between three different strategies, to detect structural changes in a graph that are of interest for conflict detection.

- **Attribute update:** The value of an attribute has been changed. E.g., The attribute ‘minimumAge’ has been set from ‘21’ to ‘18’.
- **Reference update:** The set of referenced model elements has been changed. E.g., New model elements have been added or removed. (c.f. Fig. 2)
- **Role update:** A model element is referenced or de-referenced by another model element.
- **Referenced element update:** A referenced model element has undergone an update.

For reasons of simplicity the modifications in the examples are of just two kinds, namely inserting a new *Activity* into a *Sequence*, which affects the *Sequence* through a reference update (REF), and connecting an *Activity* through a *Link*, in which case the *Activity* is affected through a role update (ROL). For both the “syntactic” as well as the “semantic” side, a set of elements (Con) that have undergone conflicting modifications is computed. For instance, depending on whether the set of syntactic conflicts is empty, a syntactic conflict has or has not been detected.

In (A), the first developer (V') adds a new *Activity* $A0$ and connects it with a *Link* $L01$ to $A1$, whereas the second developer (V'') inserts a new *Activity* $A4$ into *Sequence* $S1$. The affected model elements are $A1$ and $S1$ due to a role and a reference update. In the semantic view, $A1$ and $A3$ are affected. Therefore, neither a syntactic nor a semantic conflict occurs.



Syntax
Semantic View
M2
BPEL Metamodel
Conflict detection strategy
M1
No Syntactic & No Semantic Conflict
No Syntactic & Semantic Conflict
Syntactic & No Semantic Conflict
Syntactic & Semantic Conflict
A
B
C
D
Con = {}
Con = {A3}
Con = {S1}
Con = {A1}

Fig. 2. BPEL example

Similar to (A), two *Activities* are added in (B), which do not result in a syntactic conflict as $S1$ and $A3$ are affected. In the semantic view however, as the two new *Activities* $A4$ and $A5$, were inserted at the end of the *Sequence*, a conflict arises in $A3$. In (C), a new *Activity* $A4$ and a new *Sequence* $S2$ is added. This results in a syntactic conflict, as $S1$ is affected through a reference update. However, the adding of $S2$ does not affect execution order, and since in the semantic view both modifications are equal, no semantic conflict occurs.

Finally in (D), a syntactic conflict occurs through adding new elements to $S1$. These modifications, however, are not equal as they were in the previous scenario and affect $A1$ through a role update resulting in a semantic conflict.

The following OCL expressions define the above used conflict sets in more detail. The *Con* set contains all conflicting model elements and is a union of three further sets that represent update-update, create-create and update-delete conflicts accordingly. The set *UpdCon* consists of concurrently edited model elements, *CrCon* contains concurrently created elements that are however not equal, and *DelCon* contains concurrently updated and deleted model elements. The function *isUpdated* determines an update to a model element and the function *areNotEqual* checks for the equality (as opposed to the identity) of two model elements.

Listing 1.1. OCL Expressions

```
Con = UpdCon->union(CrCon->union(DelCon))

UpdCon = Updates'->intersection(Updates")->select(e|e.areNotEqual(V',V"))
CrCon  = Creates'->intersection(Creates")->select(e|e.areNotEqual(V',V"))
DelCon = (Updates'->intersection(Deletes"))->union(Updates"->intersection(Deletes'))

Updates'␣=␣V->select(e|e.isUpdated(V,V')
Updates" = V->select(e|e.isUpdated(V,V")

Creates'␣=␣(V'-V)
Creates" = (V"-V)

Deletes'␣=␣(V-V')
Deletes" = (V-V")
```

These above-mentioned comparison strategies that are encapsulated in the *isUpdated* method, can be applied to individual metamodel elements, which yields the advantage of being able to fine tune optimistic or pessimistic behavior of the conflict detection mechanism. For instance, in examples depicted in Fig. 2, we assumed that a model element has undergone modification if one of its references has been changed (reference update). Furthermore, we recognize an update to a model element, if a link of a reference pointing to that model element has been created or removed (role update). Such a strategy makes sense if the role of a model element that is indicated by a reference influences the meaning of a model in a way, that can result in possible conflicts. Adding new elements to a simple container-like model element would for instance not result in a conflict. It may however be useful to be informed of a conflict if a reference denoting a more specialized role is modified.

3 Prototype

After the previous section has introduced our approach from a conceptual point of view, the following section will describe our prototype application from a more technical perspective. Furthermore, as yet the given examples have abstracted from check-in/check-out functionality and simply assumed the existence of an ancestor revision and two working copies, an example describes how the prototype implementation deals with accumulating the differences between several revisions before an actual comparison takes place.

In order to define the abstract syntax of a modeling language and a desired semantic view a metamodeling architecture is needed. The "Eclipse Modeling Framework" (EMF) [11] provides Ecore, which is a simplified version of MOF

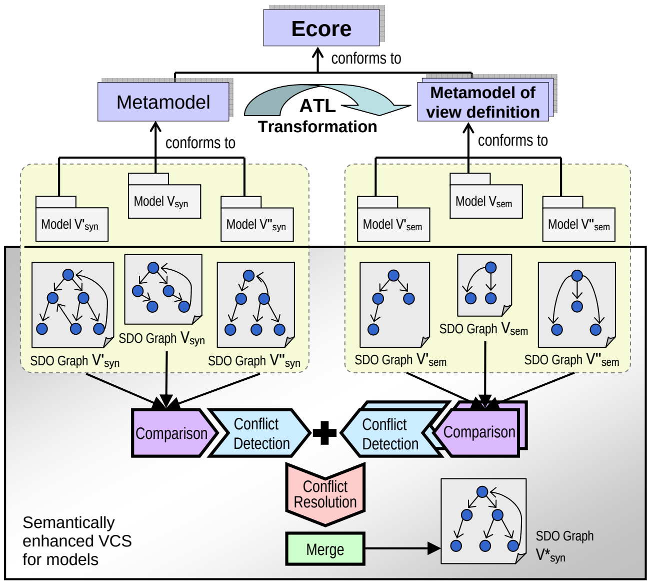
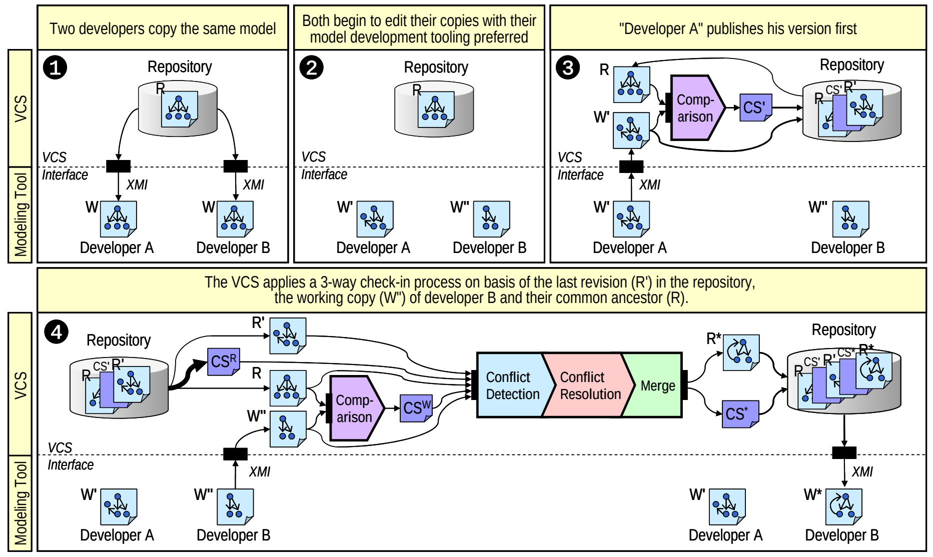


Fig. 3. Architecture of the proposed semantically enhanced VCS for models

that constitutes an M3 layer. Furthermore, EMF covers persistence support with an XMI serialization mechanism and a reflective API for manipulating EMF models. The creation of a semantic view from a model is realized through the "Atlas Transformation Language" (ATL) [12], which is a QVT-like model-to-model transformation language. Accordingly, the top of Fig. 3 shows the usage of this metamodeling stack in the context of our prototype's architecture.

The comparison of the versions ($V'_{syn}, V''_{syn}, V'_{sem}, V''_{sem}$) with their common ancestor (V_{syn}, V_{sem}) is carried out on a generic graph representation of the respective models and views. For this purpose, the EMF reference implementation of "Service Data Objects" (SDO) [13] is used. SDO is a general framework to realize standardized access to potentially heterogeneous data sources such as databases, XML files or models serialized in XMI. SDO allows to create "datagraphs" from EMF models, which are convenient for comparison purposes as SDO's mechanism to establish the difference between two graphs can be used. These so called "change summaries" are used in our prototype to store modifications between versions, which are then used by the actual conflict detection mechanism. Hence, the underlying algorithm implements the aforementioned comparison strategies and establishes the relevant sets of conflicting elements. This comparison component of our prototype is implemented in Java on top of SDO and EMF.

Figure 4 illustrates a workflow scenario of the proposed VCS including the actually so far not integrated phases conflict resolution and merge which are part of the check-in process.



Two developers copy the same model
Both begin to edit their copies with their model development tooling preferred
"Developer A" publishes his version first
VCS
Modeling Tool
VCS Interface
Repository
R
W
XMI
W'
W''
Developer A
Developer B
Comp-arison
CS'
The VCS applies a 3-way check-in process on basis of the last revision (R') in the repository, the working copy (W'') of developer B and their common ancestor (R).
CS^R
CS^W
Conflict Detection
Conflict Resolution
Merge
R*
CS*
W*

Fig. 4. Workflow scenario of the proposed approach

To start with, two developers A and B contact the repository and create a personal working copy (W) - a local reflection of the repository's file ❶. Developers then work in parallel, modifying their private copies (W' and W'') ❷. Developer A saves his changes to the repository first. Because the last revision in the repository is the direct ancestor of the incoming working copy (W') the check-in can proceed. The files saved in the repository are the modified working copy of developer A (W') and the computed change summary (CS'), provided by SDO, of W' and his ancestor (R) ❸. When developer B attempts to save his changes later, the repository informs him that his artifact (W'') is out-of-date. In other words, that artifact W'' is not a working copy of the current last revision in the repository (R'). Therefore, the VCS has to apply a 3-way check-in process containing the phases comparison, conflict detection, conflict resolution and merge. The process starts with comparing the working copy of developer B (W'') with his ancestor (R) to determine the modifications (CS^W). To be able to compute conflict detection the change summary between the last revision in the repository (R') and the common ancestor (R) has to be retrieved. In this workflow scenario there only exists one change summary between the revisions but if more then one exist they have to be accumulated resulting in CS^R. Now the conflict detection, resolution and merge process can start involving the developer B. Once the developer B has both sets of changes integrated, he saves the merged artifact (R^*) and the change summary (CS^*) back to the repository ❹. Generally, in order to work always on the actual artifact from the repository, developers have to update their current artifact to avoid unnecessary 3-way check-in processes.

4 Case Study

This section discusses an evaluation of our approach regarding its capabilities for semantic conflict detection upon three different application scenarios. Two scenarios deal with behavioral and structural modeling, whereas the third scenario deals with applying our approach to imperative programming languages. These example scenarios were chosen because on the one hand we deem them representative for various modeling languages in practical use, and on the other hand, to illustrate the flexibility of our approach not just for models, but possibly for code-oriented software artifacts, too.

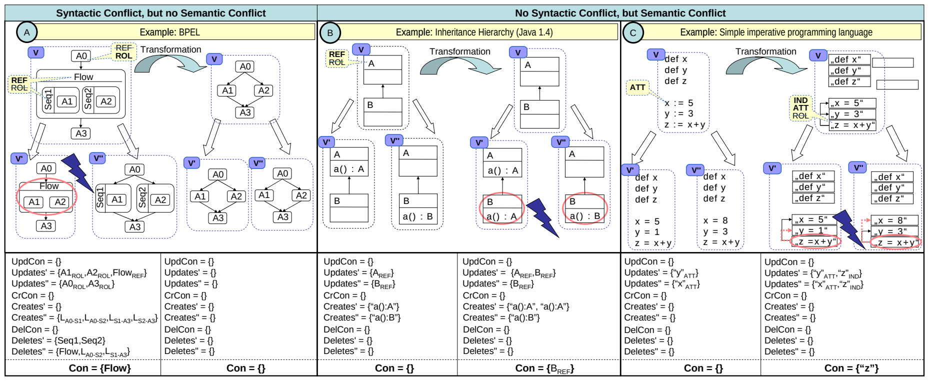
As shown on the left-hand side of Fig. 5, the first scenario (A) deals with conflict detection between BPEL documents. For our evaluation we have extended the previously introduced BPEL metamodel and the assorted semantic transformation, mainly by including the Flow construct denoting parallel execution of activities. Since basically an almost arbitrary mixture of the Sequence, Flow and explicit Link constructs can be used to model a process definition, semantic conflict detection gives considerable benefits in actually separating merely syntactic from "real" semantic conflicts. Our experiments have shown us, that without semantic conflict detection, spotting the latter becomes a tedious task as they can easily be obscured by BPEL's "syntactic sugar".

The second scenario (B) deals with the inheritance of methods in a Java class hierarchy. Thereby we aim at detecting conflicts that involve updates of inherited methods. The concept of inheritance is made explicit through a semantic view that propagates all inherited methods down the class hierarchy, which in turn allows semantic conflict detection based on the created view. Consequently, a semantic conflict can be detected, as both developers have introduced a method with the same name but different return type.

The third scenario (C) deals with semantic versioning of a simple imperative programming language. Thereby, a program is transformed into a dependency graph, which makes explicit data dependencies between statements in the code. Thus, for instance, concurrent changes to two different statements that influence some other model element can be detected. In the example shown, this is the case as the statements setting the variables x and y are modified, which indirectly updates the statement setting the value of z.

The effort for specifying the transformations for each of these examples was considerably small, as each of the above scenarios account for only about 50 to 200 lines of ATL transformation language code. Thus, the return on investment gained in better conflict detection clearly outweighs the initial effort spent on specifying the semantic views. Furthermore, the above examples emphasize the versatility of a model transformation-based approach, as one gains the ability to perform diverse tasks like eliminating syntactic sugar (A), explicate hidden concepts in models through the application of inference rules (B), and in general the establishment of specialized views on models (C) that highlight certain aspects of interest.

Concluding, we perceive that the strength of our approach lies in its way of specifying semantics through model transformations, as apposed to a rigorous



UpdCon = {}
Updates' = {A1$_{ROL}$,A2$_{ROL}$,Flow$_{REF}$}
Updates'' = {A0$_{ROL}$,A3$_{ROL}$}
CrCon = {}
Creates' = {}
Creates'' = {L$_{A0-S1}$,L$_{A0-S2}$,L$_{S1-A3}$,L$_{S2-A3}$}
DelCon = {}
Deletes' = {Seq1,Seq2}
Deletes'' = {Flow,L$_{A0-S2}$,L$_{S1-A3}$}
	UpdCon = {}
	Updates' = {}
	Updates'' = {}
	CrCon = {}
	Creates' = {}
	Creates'' = {}
	DelCon = {}
	Deletes' = {}
	Deletes'' = {}
		UpdCon = {}
		Updates' = {A$_{REF}$}
		Updates'' = {B$_{REF}$}
		CrCon = {}
		Creates' = {“a():A”}
		Creates'' = {“a():B”}
		DelCon = {}
		Deletes' = {}
		Deletes'' = {}
			UpdCon = {}
			Updates' = {A$_{REF}$,B$_{REF}$}
			Updates'' = {B$_{REF}$}
			CrCon = {}
			Creates' = {“a():A”, “a():A”}
			Creates'' = {“a():B”}
			DelCon = {}
			Deletes' = {}
			Deletes'' = {}
				UpdCon = {}
				Updates' = {“y”$_{ATT}$}
				Updates'' = {“x”$_{ATT}$}
				CrCon = {}
				Creates' = {}
				Creates'' = {}
				DelCon = {}
				Deletes' = {}
				Deletes'' = {}
					UpdCon = {}
					Updates' = {“y”$_{ATT}$,“z”$_{IND}$}
					Updates'' = {“x”$_{ATT}$,“z”$_{IND}$}
					CrCon = {}
					Creates' = {}
					Creates'' = {}
					DelCon = {}
					Deletes' = {}
					Deletes'' = {}

Con = {Flow}	**Con = {}**	**Con = {}**	**Con = {B$_{REF}$}**	**Con = {}**	**Con = {“z”}**

Fig. 5. Case study scenarios

mathematical definition of semantics, model transformations are immediately executable and besides expressing the actual semantics of the modeling language, would for instance also allow to explicate application specific rules or some form of model metrics.

5 Related Work

The closest approach to ours is laid out by *SemVersion* [14], which is itself based on RDF, proposing the separation of language specific features (e.g., semantic difference) from general features (e.g., structural difference or branch and merge). To perform the semantic difference the semantics of the used ontology language has to be taken into account. Therefore, assuming using an RDF Schema as the ontology language and two versions (A and B) of an RDFS ontology, *SemVersion* uses RDF Schema entailment on model A and B and infers all triples it can. Now, a structural difference on A and B can be calculated in order to obtain the semantic difference. In our approach semantic differences between models also composes of a transformation in the semantic representation followed by a structural difference computation. SemVersion, however, compared to the work presented in this paper is not flexible to operate on any modeling language and furthermore does not provide version control functionalities.

In terms of optimistic state-based VCS, *Odyssey-CVS* [7] presents a graph-based system for versioning UML elements, aiming to support UML-based CASE tools in evolving their artifacts. Odyssey-CVS, therefore, is not flexible in the used modeling language but open to incorporate any modeling tool. Model differences and conflicts found during the comparison and conflict detection phase are results of a purely structural comparison of two versions of a model with their common ancestor. Hence semantic aspects of a modeling language are not considered.

In terms of comparison and difference detection between versions of models *Ohst et al.* [15] addresses the problem of how to detect and visualize differences between versions of UML models such as class or object diagrams. The difference computation algorithm proposed detects only structural differences visualized in an appropriate way. Ohst et al. provides an important approach for the comparison phase, that is part of the VCS's check-in process. The approach presented is loosely-coupled to the modeling tooling but provides difference computation for UML models, only. Furthermore, Ohst et al.'s approach is not extensible in order to "understand" and interpret the semantics of a model.

More widely related to our proposed approach, are VCSs which focus on models but are tightly-coupled to the modeling environment in order to be able to save operations performed on models. For example, *Nguyen* [8] proposes a VCS which deals with the detection of structural and textual differences between versions of many kind of software artifacts, including models. *Oda and Saeki* [9] describe the need for a graph-based VCS which manages the changes on a model and its components (e.g., classes, associations and attributes). The proposed VCS, moreover, is able to handle various kind of elements that are different according to the used diagrams (UML, ER). Oda and Saeki's approach does not support concurrent editing and therefore is not supporting a check-in process which is capable to merge different model versions. Overall, beside the fact that semantics of the modeling languages are not considered, these works are accurate since structural changes from editors are stored but therefore close interoperability between an editor and the VCS repository is needed.

6 Conclusion and Future Work

In this paper proposes a light-weight approach for incorporating semantics into VCS for models. It is argued that by means of semantics conflicts between concurrently edited model version can be found more precisely since syntactic sugar can be eliminated resembling the same concept and hidden concepts can be explicated. For this semantics views generated by model transformations conflicts between model version and standard graph-based model comparision is employed on the abstract syntax as well on the semantic view thus providing for extensibility. The approach, for which a prototype is presented, which is open to the modeling environment through using XMI as an exchange format between the VCS and the modeling tooling. Furthermore, it relies on meta-modeling techniques and MDD standards thus provide for flexibility of the approach to operate on virtually any modeling language. The benefits of the proposed approach are exemplified through three diverse scenarios revealing that with a relatively small amount of effort to establish the necessary transformations, a return on investment can quite easily be gained.

With respect to future work, the approach presented has the potential to define not only one semantic view but multiple semantic views which may focus on cerain semantic aspects each and thus promises increase conflict detection precision. Future research needs to elaborate on how to extend the conflict detection approach to also operate on multiple semantic views. Also worthwile is the investigation into the conflict detection strategies may be specifically fine-tuned towards different modelling languages and whether some general guidelines can be derived. At the current state of development the approach solely focuses on the phases comparison and conflict detection. Quesions how these conflicts can be visualized for the developer and how they can be resolved and merged is going beyond the scope of this paper. Therefore in the future we will investigate how the semantics incorporated can also improve the conflict resolution and consecutive model merging phase to semantically enhance all phases of the CVS.

References

1. Subversion. (http://subversion.tigris.org/)
2. Concurrent Versions System. (http://www.nongnu.org/cvs/)
3. Bendix, L., Larsen, P.N., Nielsen, A.I., Petersen, J.L.S.: CoEd – A Tool for Versioning of Hierarchical Documents. In: ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management. Volume 1439 of LNCS., Springer (1998) 174–187
4. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering **28** (2002) 449–462
5. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? Computer **37** (2004) 64–72
6. Slonneger, K., Slonneger, K., Kurtz, B.: Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
7. Oliveira, H., Murta, L., Werner, C.: Odyssey-VCS: a Flexible Version Control System for UML Model Elements. In: SCM '05: Proceedings of the 12th international workshop on Software configuration management, ACM Press (2005) 1–16
8. Nguyen, T.N.: A Novel Structure-Oriented Difference Approach for Software Artifacts. In: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC). Number ISBN: 0-7695-2655-1, IEEE Computer Society (2006) 197–204
9. Oda, T., Saeki, M.: Generative Technique of Version Control Systems for Software Diagrams. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE Computer Society (2005) 515–524
10. Business Process Execution Language for Web Services version 1.1: Specification. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ (2007)
11. EMF Homepage. http://www.eclipse.org/modeling/emf/ (2007)
12. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL – Eclipse Support for Model Transformation. In: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France. (2007)
13. SDO Homepage. http://www.eclipse.org/modeling/emf/?project=sdo (2007)
14. Völkel, M.: D2.3.3.v2 SemVersion – Versioning RDF and Ontologies. $http : //www.aifb.uni - karlsruhe.de/Publikationen/showPublikation?publ_id = 1163$ (2006)
15. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: Proceedings of the 9th European Software Engineering Conference (ESEC). Number ISBN: 1-58113-743-5, ACM Press (2003) 227–236