Why Model Versioning Research is Needed!? An Experience Report*

Kerstin Altmanninger[2], Petra Brosch[1]**, Gerti Kappel[1], Philip Langer[2], Martina Seidl[1], Konrad Wieland[1], and Manuel Wimmer[1]

[1] Business Informatics Group, Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Department of Telecooperation, Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

Abstract. The status of current model-driven engineering technologies has matured over the last years whereas the infrastructure supporting model management is still in its infancy. Infrastructural means include version control systems, which are successfully used for the management of textual artifacts like source code. Unfortunately, they are only limited suitable for models. Consequently, dedicated solutions emerge. These approaches are currently hard to compare, because no common quality measure has been established yet and no structured test cases are available. In this paper, we analyze the challenges coming along with merging different versions of one model and derive a first categorization of typical changes and the therefrom resulting conflicts. On this basis we create a set of test cases on which we apply state-of-the-art versioning systems and report our experiences.

Key words: model versioning, conflict categorization, tool evaluation

1 Introduction

With the increasing employment of model-driven engineering techniques for software development, the call for adequate infrastructural means supporting the effective management of software models grows ever louder. Tools successfully used for versioning textual artifacts like source code are only limited suitable for models, due to their line-oriented text comparison component.

The urgent need for a suitable infrastructure supporting effective model versioning has been widely recognized and first solutions start to emerge (cf. [3] for a survey and [1, 16, 18, 20] for model versioning approaches). In this young research area we are currently in such an early phase that even the research questions are not clearly stated and that goals and borders are not clearly defined, going seldom beyond the demand for precise conflict detection and supportive conflict

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584.
** Funding for this research was provided by the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.

resolution. Furthermore, no established quality criteria are available. Hence, the evaluation and comparison of model versioning systems in a structured and comprehensible manner are hardly possible. Related work to this paper only focuses on particular phases of the versioning process as it is done in [10, 14]. Recently, Barett et al. [4] have performed an evaluation of the versioning capabilities of commercial modeling tools and provide an experience report. In this paper, we follow a similar approach with the difference that we start from a discussion of the phases passed through the merge process. This allows us to create a structured test set and categorize the open issues and problems of model versioning.

The paper is organized as follows. In Section 2 we propose a criteria catalogue consisting of test cases for the evaluation of model versioning systems. To establish a comprehensive test set, we analyze and categorize the phases run through during the merge of diverging working copies of one model. In Section 3 we apply state-of-the-art model versioning systems on the test cases in order to devise requirements and challenges for model versioning in Section 4 before we conclude the paper in Section 5.

2 Criteria Catalog for the Evaluation of Model Versioning Systems

In this paper, we aim at establishing test cases for the structured and repeatable evaluation of model version control systems. The test cases emerge from model versioning scenarios potentially leading to conflicts in order to test change detection and conflict detection facilities at various complexity levels. These scenarios are systematically categorized according to the phases of the versioning process and the different dimensions of the model merge problem. Due to space limitations we only discuss the categorization briefly and give some representative examples. For a complete description of the criteria catalog we kindly refer to our project homepage[1].

The most general layer of the categorization is set up according to the three phases of the versioning process, which are depicted on the right side of Figure 1. In the *change detection* phase the performed modifications on two working copies of one model are identified. The detected changes build the basis for the two subsequent phases, the *conflict detection* and *inconsistency* detection. In the second phase conflicts are detected by analyzing *concurrent changes* solely, whereas in the third phase *consistency problems* are revealed which would occur in the merged model incorporating all changes.

Change Detection. In this phase changes performed in parallel on the common base revision *V0* resulting in the modified versions *V0'* and *V0"* are identified (cf. Δ in Figure 1). The change detection may be realized either in a state-based manner which considers only the final states of the modified versions or by a change-based approach where the modeling editor tracks the executed

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

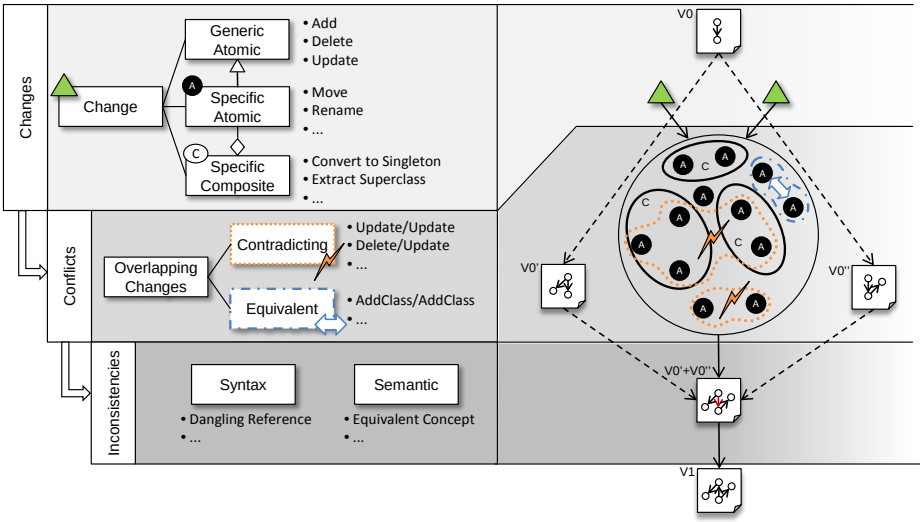

Fig. 1. Conflict categorization according to the phases of the versioning process.

operations directly [19]. However, the quality of the change detection directly correlates with the quality of the merged version.

We classify changes according to two orthogonal dimensions. The first dimension represents their dependency on an underlying modeling language. A change is *generic* if it may be applied to any model irrespectively of the modeling language. In contrast, a *specific* change depends on a certain metamodel. Consequently, a specific change is a specialization of a generic change. The second dimension considers the divisibility of an operation by distinguishing between *atomic* and *composite* changes.

From the resulting four combinations of these classifications, we omit *generic composite* because a composite operation makes only sense in the context of a specific modeling language. *Generic atomic changes* comprise the primitive atomic operations `add`, `delete`, and `update`. They may be performed on model elements (`add`, `delete`) and model properties (`update`) independently of the modeling language, i.e., the underlying metamodel. Generic atomic operations build the basis for more complex and language specific operations. *Specific atomic changes* are indivisible language-dependent operations like `rename` and `move`. The operation `rename` modifies a specific property which assigns—according to the underlying metamodel—a name to a model element. The operation `move` changes the containment of a model element which also requires knowledge on the underlying metamodel. Note that in certain environments `move` is realized as composite operation if allowed by the metamodel. The detection of *specific composite changes* like refactorings is challenging, but extremely important for the quality of the overall merge result [8, 12]. It enables a more compact representation of the difference report by folding atomic operations which belong to

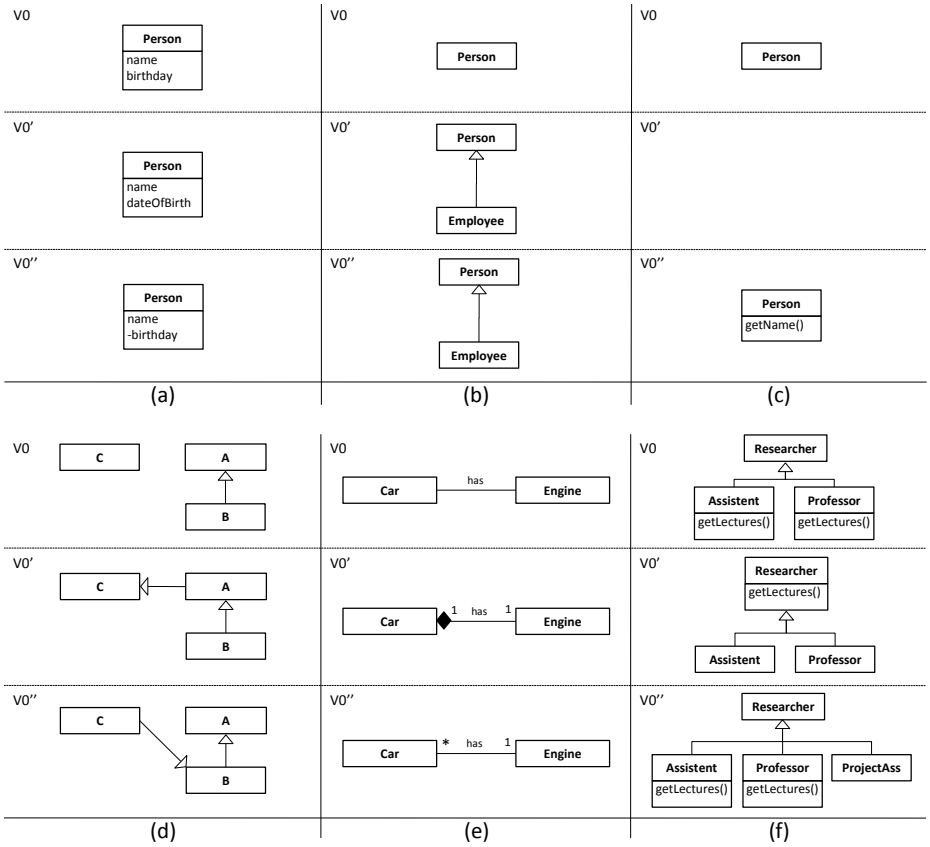

Fig. 2. (a) Unit of comparison (b) Add same class (c) Delete class vs. add method (d) Inheritance cycle (e) Contradiction in reference (f) Inheritance of methods

a composite operation. Thus, detecting applied composite operations allows a faster and better understanding of the modeler's original intention. Furthermore, it facilitates smarter conflict detection and resolution leading to a reduction of conflict alerts and identification of otherwise unrevealed merge issues.

The detection of the changes yields the basis for the conflict detection. Conflicts potentially occur whenever the modifications performed by two different modelers are overlapping, i.e., if the same model elements are involved. In some cases overlapping changes might not result in conflicts at all. Figure 2 (a) illustrates a test scenario in the domain of UML Class Diagrams where both modelers modify different properties, namely the visibility and the name, of the same attribute. It depends on the unit of comparison whether this situation ends up with a conflict or not. If the unit of comparison is the class or the attribute, then a conflict occurs. If the unit of comparison is more fine grained and the proper-

ties of the attribute elements are observed independently, no conflict should be reported.

Conflicts. Conflicts arise if parallel changes are overlapping within the same unit of comparison. Overlapping changes may either be *equivalent* or *contradicting*.

As indicated by its name, for *equivalent overlapping changes* two modelers perform the same modifications or different modifications with the same impact. Although such operations are overlapping, i.e., changing the same parts of the common base model, no conflict should be reported and only the modifications performed in one working copy should be included in the merged model. Figure 2 (b) illustrates a test scenario where both modelers add an equally named class inheriting from the same superclass.

Contradicting overlapping changes are concurrent modifications that do not commutate, i.e., their execution order affects the result [17]. Consequently, the operations cannot be merged and a conflict occurs. Such conflicts are caused by two concurrent `update` changes performed on the same property. Another kind of contradicting overlapping changes are concurrent `delete` and `update` operations involving the same model elements and properties as depicted in Figure 2 (c). Resolution of `update`/`update` conflicts as well as `delete`/`update` conflicts usually require manual interaction.

Inconsistencies. Finally, problems may occur regarding the consistency in the merged version of the working copies. According to the classification of Mens [19] we group such problems based on the level of language-specific representation.

If the merged model is not conforming to the metamodel or violates any other validation rules, a *syntactic problem* should be reported. Obviously, language-specific information, i.e., the metamodel and validation rules, has to be regarded to enable the detection of such problems [5]. Two examples of such a problem are shown in Figure 2 (d) and (e). In Figure 2 (d) both modelers add an inheritance relationship between the same classes but with different directions resulting in an inheritance cycle when merged. In the second example one modeler turns an association into a composition whereas the other increases the multiplicity at the same association end. Given these scenarios, a naive merge simply combines both modifications, resulting in an invalid model.

Although a merged model is valid, i.e., no syntactic problems are at hand, several issues may exist with respect to the static and/or operational semantics. Since the semantics and the correct interpretation of a model is difficult to express in a formal way, the detection of such problems is challenging. For instance, if two modelers express the same concept in linguistically different ways, a semantic-aware merge may prohibit an undesired merge result in which the same concept is contained twice. An example is given in Figure 2 (f) where one modeler shifts a method contained in two classes into their common superclass whereas the other modeler introduces a new subclass. Consequently, the new class inherits also the shifted method which might not be intended.

3 Evaluation of Model Versioning Systems

The categorization given in the previous section allows us to create test cases for the evaluation of conflict detection components in model versioning systems. In the following, we apply selected versioning systems on our test cases and discuss the results of the experiment.

Selected Versioning Systems. In previous work [3] we have conducted a survey on the state of the art of three-way merging approaches for model versioning systems. On basis of this survey, we have selected a set of versioning systems consisting of a text-based tool, a commercial tool, an open-source tool, and a tool developed in the context of a research project.

We have evaluated *Subversion*[2] as representative of line-based versioning systems for text-based artifacts and three solutions dedicated to set model artifacts under version control. The IBM *Rational Software Architect* (RSA)[3], a UML based model-driven development tool, provides a series of model management operations, including comparison and merge functions. The Eclipse plug-in *EMF Compare* [9] allows matching, comparing, and merging Ecore-based models. Finally, the CASE tool *Unicase* [13] provides a repository which is under version control. The Unicase client allows viewing and editing models in a textual, tabular, or graphical representation. The comparison algorithm uses the editing operations obtained from the Unicase client.

Experimental Setup. For this evaluation, we have focused on versioning UML Class Diagrams as this language is supported by all tools. The test cases cover all previously identified conflict categories. Due to space limitations we kindly refer to the AMOR project website[4] for a detailed description.

To compare the quality of the conflict detection results of the selected tools, we reuse measures stemming from the field of information retrieval to compare the manually determined conflicts (the "relevant conflicts") to the automatically found conflicts. The primary measures are *precision* and *recall* which are negatively correlated. Thus, we use a common combination of the primary measures, namely the *F-measure*. The measures are based on the notion of *true-positives* (tp), *false-positives* (fp), *false-negatives* (fn), and *true-negatives* (tn). In our evaluation we strictly rated the outcome in comparison to the expected result which we specified for each test case.

Results. Table 1 reports the results of our evaluation for representative test cases, summarized by the precision, recall, and F-measure values. The highlighted fields indicate that although other conflicts have been reported the expected result has not been accomplished. The precision values reflect the importance of operating on adequate representations of the graph-based models (cf.

[2] `http://subversion.tigris.org/`
[3] `http://www-01.ibm.com/software/awdtools/architect/swarchitect/`
[4] `http://www.modelversioning.org/`

Table 1. Evaluation results.

Category	Examples	Subversion	RSA	EMF Comp.	Unicase
No Conflict	Add Different Class	fp	tn	tn	fp
	Add Different Reference	fp	tn	tn	fp
	Unit of Comparison	fp	tn	fp	tn
	Rename vs. Move	fp	tn	fp	tn
Contradicting	Rename Class	tp	tp	tp	tp
	Delete Rename Class	tp	tp	fn	tp
	Delete Class vs. Add Method	tp	tp	tp	tp
	Containment	fn	fn	tp	fn
Equivalent	Add Same Class	fn	fn	tp	fn
	Contradiction in Hierarchy	fn	fn	fn	fn
Syntax	Dangling Reference	fn	tp	tp	tp
	Inheritance Cycle	fn	tp	fn	fn
	Contradiction in Reference	fn	tp	fn	fn
Semantic	Inheritance of Methods	fn	fn	fn	fn
	Semantics in Associations	fn	fn	fn	fn
Measures	Precision	0.43	1	0.71	0.67
	Recall	0.27	0.55	0.45	0.36
	F-Measure	0.34	0.71	0.55	0.47

Subversion vs. RSA). In contrast to the high precision values, the recall values are less promising, indicating that certain conflicts are not automatically detectable at all influencing the F-measures values. In the following we report our experiences with the different tools.

Subversion. We used Subversion to put the serialized models under version control. This has been directly done in the file system independently of the modeling editor. For the conflict resolution we have used the default text-based tool which marks and shows the location of the conflicting changes in the files. First, we have tried to version XMI-serializations of models created with the Enterprise Architect (EA). Since the EA includes much additional information like on graphical representation and time stamps in the XMI-files, many conflicts have been reported which are not related to our artificially provoked conflicts. Hence, the application of Subversion has been impractical for this kind of serialization.

Second, we have tried to version XMI-files created with the Eclipse UML 2.1 plug-in[5] which serializes the models in two files: one containing the actual model information and one containing information on the graphical representation. Due to the very generic comparison algorithm, the conflict report confines itself on indicating modifications occurring at the same location in the text file. For example, in the "Add Different Class" test case, a conflict is reported, as the new classes are inserted at the same position. Also in the "Add Same Class" test case a conflict is reported, as the inserted classes have different IDs. If not the same lines are affected, the merge is performed without any further inquiry. This may result in invalid models which, e.g., contain compositions with a multiplicity

[5] http://www.eclipse.org/uml2

higher than one or dangling references. Subversion naturally provides no specific validation mechanisms.

Overall, using Subversion showed up as being as expected, i.e., even for small examples the conflict resolution is challenging and error-prone due to missing tool support.

Rational Software Architect (RSA). The "Compare with each other" command of the RSA enables the comparison between three UML models where one of them has to be declared as ancestor. The differences of the models V0' and V0" are presented in separate windows and a preview of the merged version is provided. The RSA also reports changes concerning the graphical representation of the model.

The RSA supports the merging process in a semi-automatic manner because manual interaction is always required even if no conflict is reported. When conflicts occur, the user must decide which change—either from the left or from the right model—to perform first. Obviously, the resolution order influences the result. Furthermore, the RSA also provides the two language-specific operations *rename* and *move*. Thus, contradicting changes caused by these operations are handled without any problems. After resolving all conflicts the user has to overwrite either the left or the right working copy. The RSA also provides a validation mechanism to detect consistency problems concerning the syntax of the merged model. Unfortunately, the user may only validate the new merged version of the model after overwriting one of the working copies.

Overall, equivalent changes like "Add Same Class" are not identified as conflicts; thus, both classes exist in the merged version. The same is true for consistency problems concerning the semantic of models. The RSA detects contradicting changes in a fine-grained manner, but only if dedicated, language-specific rules exist.

EMF Compare. For using the three-way merge of EMF Compare three different files of EMF-based models need to be selected from the Eclipse workspace. With the command "Compare with each other" and the according selection of the origin model the differences and conflicts between the origin model and two modified versions are visualized in a tree-based manner. Additionally to the tree-based presentation, the differences and conflicts are visualized between the edited model versions (V0' and V0") with colors (blue for changes and yellow for conflicts). To merge the parallel evolved models changes can be "copied" from the left model artifact (V') to the right one (V") and vice versa. A special option exists to just copy the non-conflicting changes.

As stated in Table 1, about half of the test cases result in accurate conflict detection reports. Since EMF Compare does not incorporate the metamodel of a modeling language nor offers language-specific extensions, it is not able to consider language-specific constructs (e.g., inheritance, constraints, refactorings). Hence, especially for the test cases belonging to the categories syntax and semantics EMF Compare determines a series of false-negative results. Moreover, since the unit of consistency for comparison cannot be adapted according to

specific languages, false-positive results may occur like in our test cases in the "Unit of Consistency" and "Rename vs. Move" examples.

Summing up, EMF Compare offers an adequate conflict detection report for versioning any kind of EMF-based model artifacts but has naturally deficiencies in detecting language-specific conflicts.

Unicase. The CASE tool Unicase incorporates versioning support using a central repository. When checking in a model based on an outdated version, the conflict detection process is started. Although conflicts are not reported explicitly, the merge of two concurrently modified versions is intuitive and does not allow to apply conflicting changes. When merging, all changes of both sides are listed in two columns. The user may now choose which changes she likes to apply to the merged version. If the user selects a change, all conflicting changes are automatically unselected and highlighted in the opposite side. Two versions are never automatically merged without user interaction even if there is no conflict.

Changes are directly tracked while the user modifies a model. Consequently, the detection of atomic changes works precisely. However, composite changes like refactorings are not detected and as a result, they are not regarded in the conflict detection process. In general, Unicase detects `delete/update` conflicts as well as `update/update` conflicts using a fine-grained granularity (element property). Any concurrent modification of the same property may not be merged. Although this is straightforward in most of the cases, e.g., concurrent renaming of a class, it often leads to an undesired behavior. For instance, Unicase does not allow to apply all changes of both sides if two classes have been concurrently added in the same package because the containment property of the package has been updated on both sides. Unfortunately, this has often resulted in *false-positive*s in our evaluation even though Unicase generally provides an accurate conflict detection in the first two categories. Language-specific conflicts like syntactic problems have never been reported. Even a manual validation after the merge has not reported validation problems which shows that the metamodel used by Unicase does not seem to offer detailed validation rules.

To sum up, Unicase offers an adequate conflict detection on a metamodel level. Generic atomic changes and their potential contradictions are detected correctly. However, specific composite changes and conflicts which require modeling language specific knowledge remain unrevealed.

4 Lessons Learned

As the evaluation results show, unreliable tool support for model versioning forms an insurmountable obstacle for the professional application of MDE. In the following, found issues and lessons learned from the evaluation are summarized and research questions to be answered for tackling those problems are stated.

Benchmark Availability. To the best of our knowledge, there is currently no common benchmark for model versioning systems available in the related work. Hence, neither detailed requirements nor the expected run-time behavior

of such systems is specified yet. Only very high-level requirements analyses are presented in [6] and [7] elaborating the demand for precise conflict detection and supportive conflict resolution without establishing concrete test cases.

Unreliable Conflict Detection. A major deficiency is the unreliable conflict detection. False-positives as well as false-negatives occur regularly already in our small test cases. False-positives are often reported due to updated containment properties, since conflict detection mostly works on metamodel level. Furthermore, semantic inconsistencies are hardly detected as such problems are usually ignored. Research for answering the question "What is the expected result?" is needed for improving the overall check-in process.

Confusing Difference Report. The representation of changes in concurrently edited models differs from tool to tool. Since concurrent changes are not visualized by using the model's concrete syntax, but by presenting a list or a tree structure of atomic changes, those metamodel based difference reports are not intuitive and rather confusing for modelers. For providing the modeler a better understanding of what happened, two improvements are necessary. First, differences and conflicts should be presented on the modeler's point of view, i.e., on model level. Second, related atomic changes should be grouped as one composite change.

Aggregating atomic changes to composite changes in generic modeling environments is no trivial task because of the numerous combination possibilities. To tackle this problem, Brosch et al. [8] present an operation recording approach allowing the user to define language-specific composite operations by modeling small examples. Weber et al. [21] define composite change patterns for process-aware information systems in order to reduce the complexity of process changes while raising the level of expressiveness. Küster et al. [15] compute critical pairs of dependencies and conflicts for compound change operations in process models.

Single Diagram Support. Even if some specific diagram types (e.g., UML Class Diagram) are well supported by tools, extensive application of optimistic model versioning in MDE projects does not prove satisfactory. Current development strategies include employment of specific modeling languages for specific problem domains. Like Subversion [11] for the versioning of arbitrary text files, model versioning systems must provide support for arbitrary modeling languages. For making use of the rich semantics of the model's graph-based nature, language-specific adaptation of generic model versioning systems is a key research field as it is done by Altmanninger et al. [2]. Furthermore, conflicts involving multiple types of diagrams have to be considered.

Unreliable Conflict Resolution. We have learned from the evaluation that due to the open challenges in conflict detection any support for automatic conflict resolution is only a vision and current tools only support manual decision of approving changes from left or from right. This manual approach is not only cumbersome but error-prone as well. Unfortunately, checks for ensuring correct syntax and semantics of the merged version were seldom performed in the testbed. Thus, the consistency of the merged version is not guaranteed.

5 Conclusion

In this paper, we presented a first categorization of conflicts occurring during the check-in process in model versioning systems. On this basis we inferred numerous test cases which allowed us to conduct experiments with state-of-the-art tools. Our tests enabled us to compare the tools in a structured and fair manner. The results were far from satisfying, but promising. For the moment, we focused on the UML Class Diagram, but in future work we will extend the test set with test cases containing other kinds of diagrams like UML State Charts or Activity Diagrams. The long-term objective is to establish a comprehensive, expressive benchmark for the evaluation of model versioning systems which covers a multitude of different scenarios.

Overall, many interesting research questions are open in the young research area of model versioning, demanding for a common terminology and an exact formulation of the research goals. We are aware that the model versioning test cases presented in this paper are only a drop in the ocean. But in fact, having test cases is a corner stone for structured research and we are looking forward to investigate the open issues together with the model versioning research community.

References

1. M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.
2. K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR - Towards Adaptable Model Versioning. In *1st Int. Workshop on Model Co-Evolution and Consistency Management*, 2008.
3. K. Altmanninger, M. Seidl, and M. Wimmer. A Survey on Model Versioning Approaches. *Int. Journal of Web Information Systems*, 5(3), 2009.
4. S. Barrett, P. Chalin, and G. Butler. Model Merging Falls Short of Software Engineering Needs. In *2nd Workshop on Model-Driven Software Evolution*, 2008.
5. C. Bartelt. Consistence Preserving Model Merge in Collaborative Development Processes. In *Int. Workshop on Comparison and Versioning of Software Models*, pages 13–18. ACM, 2008.
6. L. Bendix and P. Emanuelsson. Collaborative Work with Software Models—Industrial Experience and Requirements. In *2nd Int. Conference on Model Based Systems Engineering*, pages 60–68, 2009.
7. P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *ACM SIGMOD Int. Conference on Management of Data*, pages 1–12. ACM, 2007.
8. P. Brosch, P. Langer, M. Seidl, and M. Wimmer. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Int. Workshop on Comparison and Versioning of Software Models*. IEEE, 2009.
9. C. Brun and A. Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE: The Europ. Journal for the Informatics Professional*, IX(2):29–34, 2008.

10. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *11th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.
11. B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media, July 2004.
12. D. Dig, T. N. Nguyen, K. Manzoor, and R. Johnson. MolhadoRef: A Refactoring-aware Software Configuration Management Tool. In *21st Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 732–733. ACM, 2006.
13. M. Kögel, J. Helming, and S. Seyboth. Operation-based Conflict Detection and Resolution. In *Int. Workshop on Comparison and Versioning of Software Models*, pages 43–48. IEEE, 2009.
14. D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Aalysis of Approaches to Support Model Differencing. In *Int. Workshop on Comparison and Versioning of Software Models*. IEEE, 2009.
15. J. M. Küster, C. Gerth, and G. Engels. Dependent and Conflicting Change Operations of Process Models. In *5th Europ. Conference Model Driven Architecture - Foundations and Applications*, pages 158–173. Springer, 2009.
16. Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-specific Models. *Europ. Journal on Information Systems*, 6:349–361, 2007.
17. E. Lippe and N. van Oosterom. Operation-Based Merging. In *5th ACM SIGSOFT Symposium on Software Development Env.*, pages 78–87. ACM, 1992.
18. A. Lucia, F. Fasano, G. Scanniello, and G. Tortora. Concurrent Fine-Grained Versioning of UML Models. In *Europ. Conference on Software Maintenance and Reengineering*, pages 89–98. IEEE, 2009.
19. T. Mens. A state-of-the-art Survey on Software Merging. *IEEE Transactions on Software Engineering*, pages 449–462, 2002.
20. D. Ohst, M. Welle, and U. Kelter. Differences Between Versions of UML Diagrams. In *9th Europ. Software Engineering Conference*, pages 227–236. ACM, 2003.
21. B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In *Advanced Information Systems Engineering*, pages 574–588. Springer, 2007.