Consistence Preserving Model Merge in Collaborative Development Processes

Christian Bartelt
Software Systems Engineering Group
University of Clausthal
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld
+49 5323 72-7160
christian.bartelt@tu-clausthal.de

ABSTRACT
Specification by models plays a decisive role, during the development process of complex systems. The division and concurrency of labor in teams is a further characteristic of such development. Therefore an efficient configuration and variant management of resulting documents is essential. In practice, a lot of established configuration management systems like CVS and Subversion are available which provide a text based merge. Unfortunately these systems are inappropriate for the management of models because they ignore the syntactic and semantic structure which is specified by the associated meta-models. Especially during the merge of model versions the mentioned systems fails at the generation of a meta-model consistent model. In this paper a proposal is presented which is enabled to merge model versions in a model based, collaborative development process. Furthermore meta-model independent methods for consistent-receiving model merging are explained.

Categories and Subject Descriptors
D.2.7 **[Software Engineering]:** Distribution, Maintenance, and Enhancement - *Version control*

General Terms
Algorithms, Management

Keywords
collaborative development, inconsistency management, model driven engineering, model merge, model versioning, software configuration management

1. INTRODUCTION
Model Driven Engineering (MDE) approaches are widely-discussed in the scientific community and are becoming more and more popular in industrial practice. MDE is a software development approach which focuses on models, as opposed to source code. But similar to source code, models are subject to a continuing evolution. In industrial practice not even one product specification is developed, but rather a lot of versions, configurations or variants are created and account for evolution. The management of evolving models is an ambitious challenge. On the one hand large models are developed concurrently by a lot of modelers in large teams and on the other hand the customization of products requires the development of several products and model variants respectively. Exemplary, four development scenarios should be illustrated in the following:

(a Sequential Enhancement) One developer works exclusively on one model in a repository workspace. This workspace gets initialized with an initial model version.

(b Concurrent Enhancement) Multiple developers enhance the same model in multiple workspaces. Each developer (or sub team) initializes a workspace with the same (initial) model version and makes modification to this model according to (a). After each developer (or sub team) has finished his work, the resulting models have to be integrated (merged) into one valid and consistent model.

(c Multiple development lines) Consider the following example: A software company has two development lines of one product. One is the maintenance of version 1.0 and the other one is the development of the new version 2.0. It turns out, that a bug that was fixed in version 1.0 (in the maintenance development line) has also to be fixed in version 2.0. An example of this use case is how to integrate the bug fix made in one development line into the other development line. In this context, a set of explicit changes respectively change sets has to be modelled as an entity that can be applied to different input models.

(d Management of product variants) Typical variants of a software project might support a certain range of functions {personal edition, professional edition} and a specific operating system {linux, windows}. Consider a software project that provides these variants. To manage the complexity of the different variants, the product has a core model that is basically the same in all variants. Separate branches {personal edition branch, professional edition branch, linux branch, windows branch} that were initialized with this model, implement the single aspects of the supported variants. To create the model of a variant x of the product, two of these branches have to be integrated into a valid and consistent model.

CVSM'08, May 17, 2008, Leipzig, Germany.

One fundamental challenge in the scenarios (b)-(d) is the synchronization of concurrent developed model versions respectively variants. The most software configuration management [3] (SCM) systems provide services like diff or merge. But the popular SCM-systems (RCS[17], CVS [4], Subversion[15]) can manage just a directory tree of text files (source code). These systems do not use the formal structure of managed content for correct diff or merge results concerning syntax and semantics. However there are some theoretical approaches of syntactic and semantic software merging [7] and model merging [1][9][12][13]. Mostly discussed in the scientific community are two application areas – model merge regarding model evolution [8] in development processes and model merge/comparison regarding synchronization of views [10][11][13]. Further some software prototypes which implement merge concepts for collaborating modeling are published [14][16]. But the MDE does not provide a scientific founded and general approach for an optimistic meta-model independent model merge which regards the syntactic and semantic structure defining by the modeling language. The independency of the meta-model has one main advantage: A general merge technique has not to be adapted for each modeling language and the implementing collaboration environment can support directly basic optimistic merge principles for models. This paper describes a proposal for the consistency-receiving merge of model versions. Therefore simplified formalism to describe the evolution of model revisions will be introduced. This formalism includes a semantic definition of an optimistic merge procedure. In section 2 a concept to manage merge-causing inconsistencies in collaborate model development is described.

2. FROM DOCUMENT MERGE TO MODEL MERGE

As aforementioned the merge of model versions in a collaborative, distributed, and concurrent development process is a fundamental operation. In this process the main challenge is the consistency receiving of the merge result. The following explanations in section 2 and 3 are confined exclusively on a solution for receiving the syntactic and semantic (static semantics) consistency of the merged model. Before presenting a resolution for this problem four characteristics of the considered model development environment are summarized which are:

(a) *Local Workspace*
Each modeler works separately on his own workspace. That means he works at a local copy of a certain model version from a central repository of the whole team. Each modeler is able to change arbitrarily the model in the local workspace.

(b) *Change Logging*
The modeling tools of all modelers allow a fine granular logging of all model changes in the local workspaces as change sets or edit scripts.

(c) *Unique Identifiable Model Elements*
The modeling tools assign each model element with a world unique identifier at creation time.

(d) *Independency of a pre-selected Modeling Language*
The meta-model of the models which are developed is not pre-defined. The collaboration environment should be supported UML models as well as domain specific models.

These characteristics influence considerably the proposed merge method which is explained in following.

Models are a special kind of documents which are characterized by two fundamental properties. On the one hand the contained information items (or model elements) have relationships between themselves and constitute a certain linked structure. And on the other hand this structure is conformed to a well defined grammar respectively a meta-model.

Before a strategy for merging models is explained, an optimistic merge method to handle concurrent changed sets of information items is introduced. Therefore the following key terms are defined:

Definition 1 (*information item*). An information item $\boldsymbol{i}$ is an atomic information which is unspecified. $\boldsymbol{I}$ is the set of all information items.

Definition 2 (*document*). A document $\boldsymbol{d} \subseteq \boldsymbol{I}$ is a finite set of information items. The relationships between information items are not considered knowingly in this definition. $\boldsymbol{D=\wp(I)}$ is the set of all documents.

Definition 3 (*primitive change*). A primitive (atomic) change $\mathbf{pc} \in \mathbf{PC}$ describes a change operation on one information item. There are two types of changes: Instances of $\boldsymbol{ADD \subseteq PC}$ mean additions and instances of $\boldsymbol{DELETE \subseteq PC}$ mean deletions of information items. $\boldsymbol{PC}$ is the set of all primitive change descriptions.

Definition 4 (*changedItem*). $\boldsymbol{changedItem: PC \rightarrow I}$ is a mapping which assigns each primitive change with its information item.

Definition 5 (*revision task*). A revision task $\boldsymbol{rt \in \wp(PC)}$ with:

$$rt \in RT : (pc_1, pc_2 \in rt) \wedge changedItem(pc_1) = changedItem(pc_2) \rightarrow pc_1 = pc_2$$

It represents a further development of a document by processing the consisted primitive changes. $\boldsymbol{RT}$ is the set of all revision tasks.

Revision tasks are semantic equivalent with revisions in a version graph and can be associated with several other revision tasks as predecessors and successors:

Definition 6 (*successiveRT / previousRT*). $\boldsymbol{successiveRT}$ and $\boldsymbol{previousRT : RT \rightarrow \wp(RT)}$ are mappings which represent not reflexive, transitive relations with:

$$rt \in RT : rt_{prev} \in previousRT(rt) \rightarrow rt \in successiveRT(rt_{prev}) \text{ and}$$
$$rt \notin previousRT^{+}(rt)$$

The term $\boldsymbol{successiveRT^{+}}$ means the transitive closure.

With the revision tasks and their relations an acyclic, directed version graph is constituted.

Definition 7 (*changeState*). $\boldsymbol{changeState: I \times RT \rightarrow \{A,D,U,C\}}$ is a function with:

$$changeState(i,rt)=\begin{cases} A, & \exists pc \in rt : changedItem(pc)=i \wedge pc \in ADD \\ D, & \exists pc \in rt : changedItem(pc)=i \wedge pc \in DELETE \\ mergeState(i,rt), & else \end{cases}$$

A – means *added*, ***D*** – means *deleted*, ***U*** – means *unknown* and ***C*** – means *in conflict* with another primitive change.

The function changeState determines the last change operation on an information item in the version graph. If changeState returns C (in conflict) then the associated information item added and deleted concurrently in two previous revision tasks. This *concurrent revision conflict* must be resolved by the user before further changeStates can be determined. The conflict resolution of this kind of simple conflict is not an issue of the further explanations.

Definition 8 (*mergeState*). ***mergeState:I×RT→{A,D,U,C}*** is a function with:

$$mergeState(i,rt)=\begin{cases} U, & (previousRT(rt)=\emptyset) \vee (\forall rt_p \in previousRT(rt) : changeState(i,rt_p)=U) \\ A, & previousRT(rt)\neq\emptyset \wedge \bigcup_{rt_p \in previousRT(rt)} changeState(i,rt_p)/\{U\}=\{A\} \\ D, & reviousRT(rt)\neq\emptyset \wedge \bigcup_{rp_p \in previousRT(rt)} changeState(i,rt_p)/\{U\}=\{D\} \\ C & else \end{cases}$$

Definition 9 (*documentRevision*). ***documentRevision : RT→D*** is a function which determines the information items of a document after processing all changes of a revision task ***rt*** and its predecessors.

$$document\,\mathrm{Re}vision(rt)=\{i \mid changeState(i,rt)=A\}$$

As mentioned, the semantics of the revision task is equivalent to the classical definition of a revision [3] in a version graph. The revision task is a dynamic definition as a pendant to the static definition of revision.

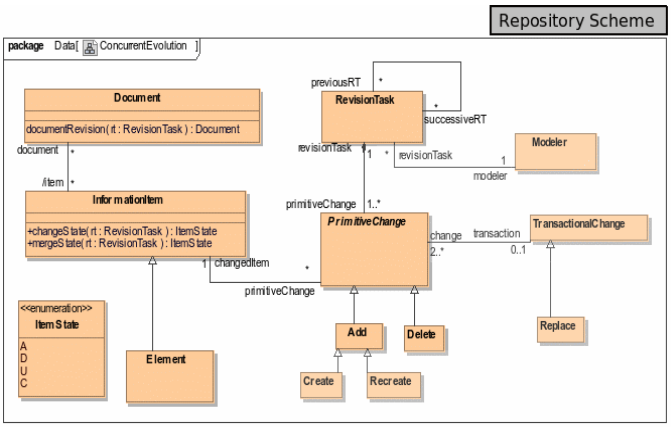

Figure 1: Simplified View on Document Evolution

This introduced merge technique implements a special kind of the known three-way merge for sets of information items which are changing concurrently. Figure 1 depicts an overview of the upper described formalism and Figure 2 illustrates this merge technique with a small example. V1, V2a, and V2b are document revisions. V1 is the origin document and V2a and V2b are revisions of V1 after further development by two different revision tasks. Note that information items in case B.6 of Figure 2 are changed by two concurrent (independent) revision tasks, but the precondition (c) at the beginning of this section requires unique identified information items. So, this case does not exist in the described merge semantics. But in semantics of evolution processes, developers duplicate changes in concurrent branches (e.g. addition of semantical equivalent information items). Therefore techniques for identification of duplicate changes are required (e.g. by using difference algorithms to compare changed document parts or model structures). Possibly the concept of SiDiff [6] is appropriated to detect of corresponding model structures in concurrent changed model versions

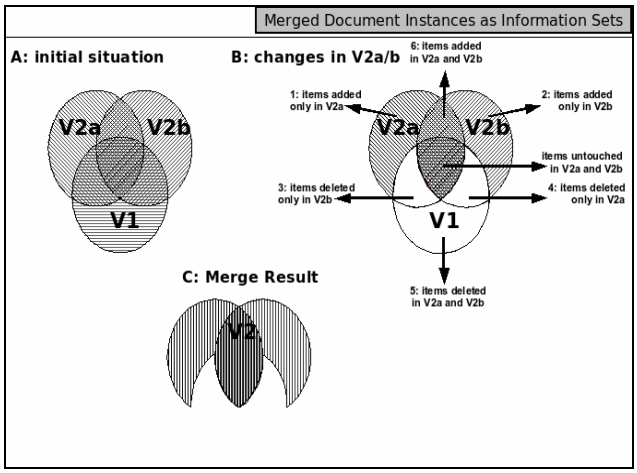

Figure 2: Example – Merge Document Revisions

The detail view B of Figure 2 illustrates the division of the primitive changes in five subsets. The concluding view C shows the resulting document revision after applying the merge technique. However models are linked structures and not sets. Their abstract syntax can be represented as a graph. How is it possible to apply the described technique of merging sets for the merging the data representation of models independently of its meta-model? To answer this question it is necessary to look at the common part of models' syntactical structure. Any kind of model consists of model elements as the most abstract entity. In UML the type "Element" is defined as root of all other specialized types. So, it is possible to define the first stage of the model merge method: The concrete information items of models are represented by instances of one abstract root type like "Element" in UML. In the following, the term *element* is only used for this mentioned root type. But the root type *element* is just abstract and a generalization of many other types in different meta-models. And furthermore the instances of element can be linked between themselves. Because of this matter of fact the primitive change types (ADD, DELETE) are not sufficed to merge linked structures. One possibility is to enhance the repository model with the further primitive change type *Replace*. For example, the dynamic model description part of XMI uses this technique (XMI differences). Therefore the merge technique must be adapted but the following examinations focus not on this issue. An interesting further topic is the consistency management of the merged model. Therefore the merged model representation has to be post-processed to get a consistent result model.

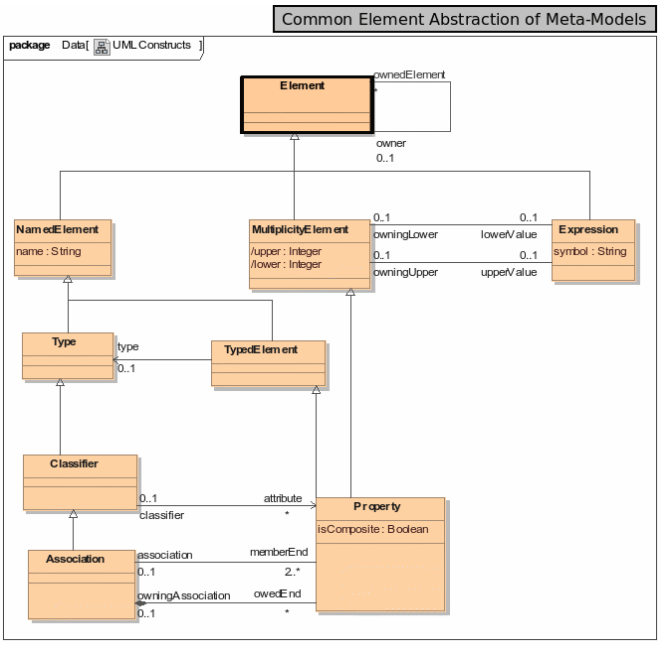

Figure 3: Example UML – Element as Most Abstract Entity

Figure 3 shows a lot of constrained relations between the meta-model data types. Additionally the UML meta-model consists further context sensitive constraints (static semantics). In the first merge stage this syntactic and semantic relations are not considered.

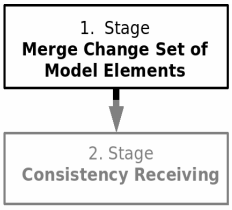

Figure 4: Merge Stages

So, the introduced merge technique handles just instances of the type Element. The management of consistency follows in a second stage (see Figure 4) which is described in the next section.

3. MERGE CONSISTENCY MANAGEMENT

Models have to be consistent to their meta-models. In general a meta-model contains a formal specification of the model syntax which can be specified exemplarily with the standardized MOF language. Furthermore it contains formal or informal descriptions about the model semantics and the concrete syntax. Constraints as context sensitive, enhanced syntax information (static semantics) can be specified exemplarily by using the standardized OCL language.

The consistency is defined as follows: A model is termed consistent if it is valid regarding to its associated formal syntax and formal static semantics specification of its meta-model. As mentioned the model merge strategy of the upper section 2 does not assure the consistency of the merged models. Figure 5 illustrates two concurrent development traces which are merged by the introduced strategy of section 2 and the inconsistent merge result. Note, Corresponding parts in both concurrent developed revisions V2a and V2b have to be identified for the determination of the merge result. As mentioned in item (c) of the environment characteristics (section 2), each versioned item has an unique identifier for tracing during the concurrent model development. The example development of Figure 5 shows some properties which are changed (e.g. upperValue modifies from 1 to 2 in V2a, aggregation modifies to #composite). However the introduced merge technique does not provide a primitive change type *REPLACE*, so that these modifications have to represent as two primitive changes *ADD* and *DELETE*. But properties are not first order entities per se. Hence, a appropriate representation of model information data whereas the mentioned properties are represented as first order entities is needed and explained in [2].

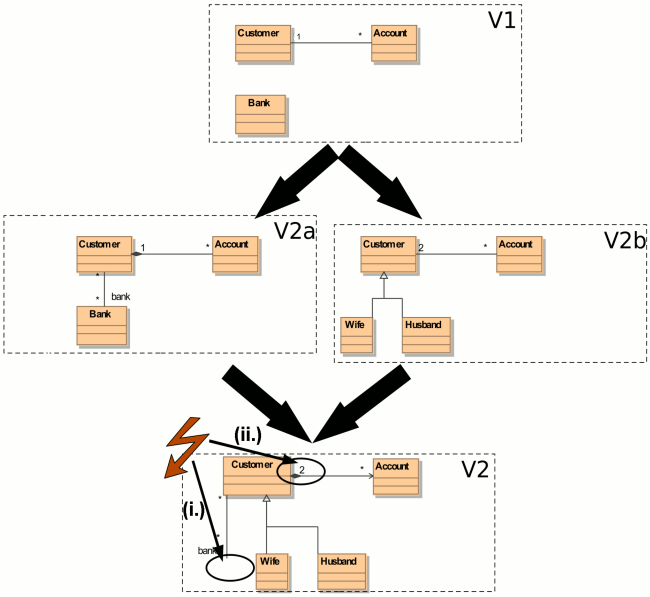

Figure 5: Inconsistencies after the First Stage of Model Merge

The inconsistency (i.) shows an association with only one end and inconsistency (ii.) shows a composite association end with multiplicity upper bound of 2.

The insufficient merge procedure of section 2 has to be enhanced with a post processing stage for consistency receiving. Consistency receiving in context of collaborated model development is including:

- Detection of inconsistencies
- Identification of inconsistency-causing model elements
- Change history investigation to determine the inconsistency-causative changes and their associated revision tasks (and modelers)
- Support for inconsistency resolving (interactive and automatic)

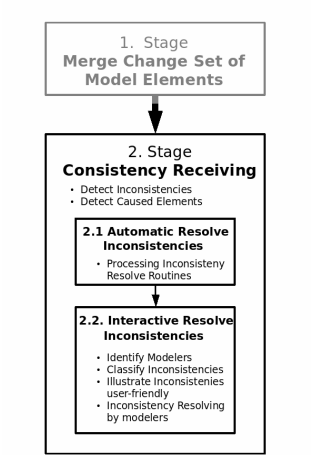

Figure 6: Second Stage – Consistency Receiving

A merge consistency conflict defines the following situation: All model versions (input e.g. V2a, V2b) which have to be merged are consistent but the merge result (output e.g. V2) contains inconsistencies (fault syntax structure). Furthermore the merge method should be universal and independent of a certain meta-model like the UML meta-model or a domain specific meta-model. In addition to the syntax specification, meta-models are often including formal described, context sensitive information. In general this is specified as several constraints between certain instances of the meta-model types (static semantics). For the description of the second stage the meta-model example is enhanced by two context sensitive constraints:

(a)
context MultiplicityElement
(self.upper->notEmpty() **and** self.lower->notEmpty()) **implies**
self.upper >= self.lower

(b)
context Property
self.isComposite **implies**
(self.upperValue->isEmpty() **or**
self.upperValue.symbol.unlimitedValue() <=1)

Both exemplary constraints are part of the UML meta-model. The detection of inconsistencies is a solved problem. It is possible to use a validation framework (e.g. EMF Validation Project) to check the formalized syntactic structure and further context sensitive constraints. The validation framework is able to list all invalid element relations and all invalid constraints.

The next step is the identification of the model elements which are causing the inconsistency. This identification method is described with the example in Figure 5.

Figure 7 depicts the relevant constraints in OCL notation for the mentioned example. Note, that the structural relation between *Association* and *Property* is also notated in OCL. The point-navigation of the OCL-Syntax is the key for the identification of inconsistency causing elements. For example, constraint (b) can be invalid by changes of instances of the regarding types *Property* and *Expression* (context *Property* / self.*upperValue*). In an instantiated model the conflicting model element instances can be evaluated. Therefore Figure 8 depicts the identification of the inconsistency and the involved element instances with the known example:

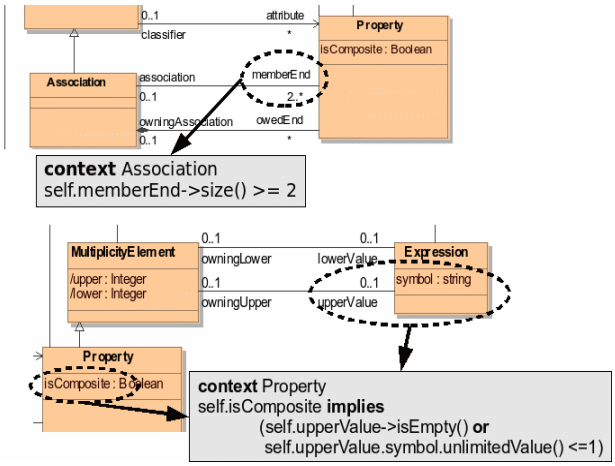

Figure 7: Identification of Inconsistencies

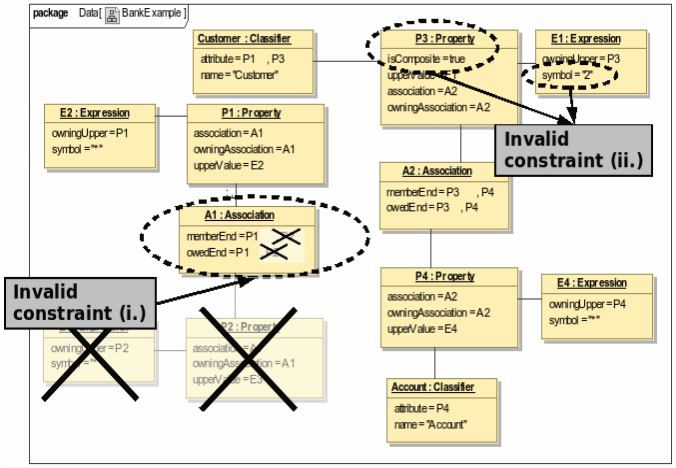

Figure 8: Identification of Inconsistency-Causing Model Elements

After the detection of the inconsistency-involved element instances, the associated primitive changes and revision tasks can be identified. The inheritance association between information item and element in Figure 1 is the relation point between the managed model and the change based model repository. Each element instance of a managed model is associated with all related primitive changes and therefore with related revision tasks and its modelers. Figure 9 shows the link relations between three model element instances (*A1*, *P2*, *Bank*) via the primitive changes *Add1*, *D1*, *D2* and the revision tasks *R-V2a*, *R-V2b* to the modelers *Wild* and *Kern*. After the identification of the modelers who were changed concurrently serval conflicting elements, the involved persons can be met and resolved the inconsistency in cooperation.

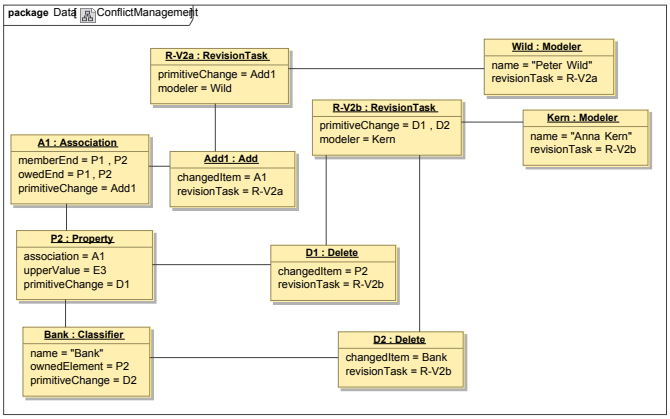

Figure 9: Identification of Involved Modelers

4. PROTOTYPE IMPLEMENTATION

The described technique of section 2 and 3 was implemented as a research prototype to specify the model merge semantics for a model management system. Therefore the repository scheme of Figure 1 is specified as an EMF-Model [5]. The functions of section 2 (e.g. changeState, mergeState etc.) were implemented as EMF-Operations and declarative specified as OCL-constraints. With the EMF code generation the JAVA-Code can be generated for this model specification including the declarative OCL-Functions. So, the instances of the repository meta-model specification can be managed by the simply generated EMF-tree editor. The Model revisions can be calculated on request by the documentRevision-function in the OCL-console of the eclipse platform. The concepts for identification of inconsistencies, involved model elements, and modelers are also implemented in the prototype. It can be requested by user in the OCL console. Currently, this change-based repository concept for model development management will be implemented on basis of the Jazz platform. Jazz supported already the collaborative management of source code but not of models.

5. SUMMARY AND FUTURE WORK

In the last sections the formal merge semantics for merging documents was introduced. This merge procedure can be adapted for merging model structures independent of their meta-model. But in general the merge definition of section 2 is insufficient for generating a meta-model consistent, resulting model. Therefore a post-processing stage is necessary to make the intermediate, resulting model of stage 1 consistent. For resolving the inconsistencies must be detected and causing model elements must be identified. If an automatic resolving of inconsistencies is possible and favored then it should be processed else the involved modelers must resolve the merge conflict. The generation of a user-friendly illustration of merge conflicts and a possibly automatic resolving of inconsistencies is not described in this paper and are going to be covered in further work.

6. ACKNOWLEDGMENTS

My thanks to Edward Fischer and Tim Schumann for helpful discussions and suggestions.

7. REFERENCES

[1] Alanen, M., Porres, I. 2003. Difference and Union of Models. In Proc. UML 2003.

[2] Bartelt, C., An Optimistic Three-way Merge Based on a Meta-Model Independent Modularization of Models to Support Concurrent Evolution. Unpublished.

[3] Conradi, R., Westfechtel, B. 1998. Version models for software configuration management. ACM Computing Surveys 30, 2, 232-282.

[4] Concurrent Versions System. http://www.nongnu.org/cvs/ (2008)

[5] EMF Homepage. http://www.eclipse.org/modeling/emf/ (2008)

[6] Kelter, U., Wehren, J., Niere, J. 2005. A Generic Difference Algorithm for UML Models, SE 2005

[7] Mens, T. 2002. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering, 28(5), p. 449-462.

[8] Mens, T., Van Der Straeten, R. 2005. On the use of formal techniques to support model evolution In: Proc. 1ères Journées sur l'Ingénierie Dirigée par les Modèles. 115-124

[9] Mens, T., Van Der Straeten, R., D'Hondt, M. 2006. Detecting and resolving model inconsistencies using transformation dependency analysis. In Model Driven Engineering Languages and Systems (p.200-214). Springer, Berlin / Heidelberg.

[10] Niu, N., Easterbrook, S., Sabetzadeh, M. 2005. A category-theoretic approach to syntactic software merging. Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on (p. 197-206)

[11] Ohst, D., Welle, M., Kelter, U.: Differences Between Versions of UML Diagrams, In: Proc. of ESEC'03, Helsinki, Finland (2003) 227–236

[12] Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W., Kotsis, G. 2007. Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In Proceedings of the 3rd Intl. Workshop on Model-Driven Enterprise Information Systems, MDEIS 2007, INSTICC PRESS, pp. 29-40

[13] Sabetzadeh, M., Easterbrook, S. 2005. An algebraic framework for merging incomplete and inconsistent views. Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on (p. 306 – 315).

[14] Scheider, C., Zündorf, A. 2007 Experiences in using Optimisitic Locking in Fujaba. Softwaretechnik Trends 27, 2 (May 2007)

[15] Subversion. http://subversion.tigris.org/ (2008)

[16] Schneider, C., Zündorf, A., Niere, J. 2004. 'CoObRA - a small step for development tools to collaborative environments'. In Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE), Edinburgh, Scotland, UK

[17] Tichy, F. W. 1985. RCS - A system for version control. Software Practice & Experience. 15. (p. 637 – 654)