An Optimistic Three-way Merge Based on a Meta-Model Independent Modularization of Models to Support Concurrent Evolution

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

Abstract—Appropriate models with a formal syntax and/or semantic are fundamental for the specification of more and more complex software systems. In practice, teams with a lot of modelers develop highly distributed and concurrently, and their models base on a common modeling language with a dedicated meta-model e.g. UML or a domain specific language (DSL). But the management and especially the synchronization (merge) of model development branches in a distributed and concurrent process is an ambitious challenge. Established configuration management systems like CVS are inappropriate to manage models because they process only text based documents and ignore the linked model structure. However the optimistic merge principle of these version and configuration management systems is popular, intuitive, and practicable. This paper explains a proposal for a meta-model independent modularization of the model representation which is appropriate to synchronize concurrent development branches with an optimistic three-way merge.

Index Terms—Modeling, Software development management

I. INTRODUCTION

Model Driven Engineering (MDE) is one of the main topics of software engineering in the scientific community and is becoming more and more popular in industrial practice. The subjects of development in MDE are specifications with well-defined models as opposed to text based documents like word documents or source code. But similar to text documents, models are subject to a continuing evolution in the development and maintenance process. 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. For the management of concurrent evolving text or source code documents, popular version- respectively software configuration management systems (SCM) like CVS [16] are established. By contrast there are only few foundations about the concurrent evolution of models.

This paper explains some aspects of the synchronization of modeling branches (modeling tasks) which are relevant to design an integrated development environment for modelers. Therefore four development scenarios should be supported by this environment and are mentioned 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. When the development task is completed, this workspace holds the final model version of the task.

(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 scenario is how to integrate the bug fix
made in one development line into the other development line.

(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.

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 (CVS [4], Subversion[14]) can manage just a directory tree of text files (e.g. 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 about 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 regarding synchronization of views [10][11]. Further some software prototypes which implement merge concepts for collaborative modeling are published [13][15]. But the MDE does not provide a scientific founded and general approach for syntactic and semantic model merge. This paper describes a proposal for an intuitive, optimistic 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 technique to manage evolving sets of information items. In section III a concept for a general modularization of model representation will be described. This modularization is necessary to apply the merge technique explained in section II.

II. FROM DOCUMENT MERGE TO MODEL MERGE

The introduction mentioned that the merge of model versions (respectively the synchronization of modeling branches) in a collaborative, distributed, and concurrent development process is a fundamental operation. The following explanations in section 2 and 3 are confined exclusively on a solution for an optimistic merge technique to manage the concurrent development of models in an integrated modeling environment. Before presenting a resolution for this problem four characteristics of the considered model development environment are summarized. They 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 universal 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 support UML models as well as domain specific models (DSLs).

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 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 i is atomic information which is unspecified. I is the set of all information items.

Definition 2 (document). A document d ∈ I is a finite set of information items. The relationships between information items are not considered knowingly in this definition. D = ∅(I) is the set of all documents.

Definition 3 (primitive change). A primitive (atomic) change pc ∈ PC describes a change operation on one information item. There are two types of changes: Instances of ADD∈PC mean additions and instances of DELETE∈PC mean deletions of information items. PC is the set of all primitive change descriptions.

Definition 4 (changedItem). changedItem: PC → I is a mapping which assigns each primitive change with its information item.

Definition 5 (revision task). A revision task rt ∈ ∅(PC) with:

rt ∈ RT := (∃pc1, pc2 ∈ rt). changeditem(pc1) = changeditem(pc2) → pc1 = pc2 represents a further development of a document by processing the consisted primitive changes. 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). successiveRT and previousRT : RT → ∅(RT) are mappings which represent
not reflexive, transitive relations with:

$rt \in \text{RT} \land rt_{\text{pre}} \in \text{previousRT}(rt) \rightarrow rt \in \text{successiveRT}(rt_{\text{pre}})$

and

$rt \notin \text{previousRT}(rt)$

The term $\text{successiveRT}$ means the transitive closure.

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

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

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


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

**Definition 8** (mergeState). $\text{mergeState}: I \times \text{RT} \rightarrow \{A, D, U, C\}$ is a function with:

$\text{mergeState}(i, rt) = \begin{cases} U, & (\text{previousRT}(rt) \cap I) \cup (\text{previousRT}(rt) \cap C) = \emptyset \\ A, & \text{previousRT}(rt) \cap I \neq \emptyset \land \bigcup_{rt_{\text{pre}}} \text{changeState}(i, rt_{\text{pre}}) = \{4\} \\ D, & \text{previousRT}(rt) \cap I \neq \emptyset \land \bigcup_{rt_{\text{pre}}} \text{changeState}(i, rt_{\text{pre}}) = \{2\} \\ C, & \text{else} \end{cases}$

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

$\text{documentRevision}(rt) = \{i | \text{changeState}(i, rt) = A\}$

As mentioned, the semantics of the revision task is equivalent to the classical definition of a revision [2] in a version graph. The revision task consists of changes and is a dynamic definition as a pendant to the static definition of revision.

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 above 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 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 semantically equivalent information items). Therefore techniques for identification of duplicate changes are required (e.g. by using difference algorithms to compare (independently of the unique identified information items) changed document parts or model structures). Possibly the concepts of SDiff [5] are appropriate to detect corresponding model structures in concurrently changed model versions.

![Figure 2: Example – Merge Document Revisions](image)

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 merging the abstract syntax of models independently of its meta-model? To answer this question it is necessary to look at the representation of models as shown in section III.

One main difference between the mentioned sets and models is that the information items of models are referenced between themselves. Furthermore the references are represented by properties which are not first order entities in most model data representations. These properties have to be changeable during the model evolution. Because of this matter of fact the primitive change types (ADD, DELETE) may be not sufficed to merge linked structures. One solution (which is used by the most popular approaches) is to enhance the change operations.
with a further primitive change type REPLACE. For example, the dynamic model description part of XMI uses this technique (XMI differences). But the mentioned merge technique with only two change types (ADD, DELETE) is intuitive, clear describable and hence attractive to adapt for model evolution management. For this reason an appropriate representation of the model structure information is explained in section III.

After an optimistic merge processing, the resulting intermediate model is not necessary consistent to the associated meta-model. So, a further interesting topic is the consistency management of optimistically merged model representations. This potential inconsistent model has to be post-processed to get a valid result model. But this consistency management is not a topic of this paper and is described in [2]. Some further discussions to model consistency management are explained in [8][9].

III. MODULES OF MODEL REPRESENTATION STRUCTURE

In the last section a certain technique for the three-way merge of concurrent developed sets of information items was introduced. But models cannot be represented by sets of information items. Models are linked structures and associated with a certain modeling language which defines the formal syntax and/or semantics. This section explains a proposal for a modular and meta-model independent representation of models which can be managed by the three-way merge technique in section II.

For a description of a meta-model independent representation of models, a discussion about the common characteristics of all models independent of their modeling language is expedient. In this paper the common characteristics of models are reduced to three fundamental properties.

(a) Models consist of model elements
(b) Model elements are related
(c) Model elements are able to contain value specifications
(d) Model elements are associated with one type of a meta-model

The two first properties (a) and (b) follows the set-relation paradigm to represent information in general. Item (c) mentions the representation of primitive data types like strings or integers etc. by model elements whereas the instance-of relation mentioned by property (d) associates syntactic and semantic constraints and descriptions of a certain meta-model with each model element. One formal foundation of the mentioned structures is the theoretical concept of typed graphs. But the merge technique described in section II is inapplicable to manage typed graphs because it deals with only one item type (information item) and provides no replacement operation. However, the management of typed graphs needs less then two item types (nodes and edges) and a replacement operation to modify the targets of the edge. This strategy is used in the majority of approaches like [15]. An enhancement of the formal three-way merge description explained in section II with an additional primitive operation replace is not suitable for an uncomplex described formalism. But if it is possible to describe the model structures with only one item concept then the introduced merge technique is applicable. In the following a module concept will be introduced which is able to represent model structures with only one manageable item type. Figure 3 depicts an overview about the model representation module.

![Figure 3: Structure to Represent Model Information](image)

Each information item is either a modeling item, an item property, or a type information item. Each modeling item represents one instance of a type from the modeling language specific meta-model. If the type is a primitive data type then the attribute symbol holds the value specification. Each item property object represents an instantiated property of the associated modeling item, a so called slot. An item property has to link the meta-model property. The entity type information item is a placeholder for an assignment between a modeling item and its type in the language specific meta-model (like the instanceOf relation). The coupling objects play a special role. These objects guarantee that information items cannot reference directly other information items. That is important to avoid open reference ends after the application of a DELETE operation on a referenced item. A reference between information items is represented only if two information items link the same coupling object. The coupling objects are not information items which are transparent and changeable directly by the model developer. They are managed internally by the system. After the description of the entities in Figure 3 the relations can be explained:

(a) Modeling items may own properties. This is represented by the two references propertyCoupling and modelingItemCoupling. This affiliation constitutes if both reference instances link the same coupling instance.

(b) Modeling items can reference other model elements or be referenced by them. This is illustrated in Figure 3 by the referenceTargetCoupling and the referenceSourceCoupling references.
another one M12, an item property instance IP1 has to be associated with M11 that links the same coupling instance with referenceTargetCoupling like M2 does with referenceSourceCoupling.

(c) Modeling items are associated with a type of the language-specific meta-model. This association is constituted analogously by indirect referencing with help of coupling objects.

Figure 4: Overview Toy Example – Library

To illustrate the application of this module concept a small toy example is presented in Figure 4. Because of its popularity UML as one example of a modeling language is used here and depicted by Figure 5. The data model contains two UML classes Library and Book which are associated. The key question is now: How is it possible to represent the small example model to manage it with the merge technique from section II? The answer is given by the upper described model data representation.

To illustrate this, Figure 6 shows an instance of the meta-model described by Figure 3 for the representation of the library example whereby the objects CL1 and CL2 represent the classes Book and Library. This model representation is the starting point of the further explanations about the concurrent evolution of the example model (applying only ADD and DELETE operations). So the concurrent modeling tasks can be synchronized (respectively merged) with the technique of section II. For the illustration of the concurrent model evolution, the example model of Figure 4 should be changed. Therefore a revision task (A) renamed the entity Book to PrintMedia. A second revision task (B) concurrently developed to A, enhanced the same entity Book with the new attribute author. The intuitive result of an optimistic merge is a
new revision which has processed both changes from A and B. Figure 8 shows an instance diagram which depicts the primitive changes Add1..Add8, Del1,Del2 of the both revision tasks A and B. All other information items like CL2, IP6, etc. are added in one or more revision tasks before starting A and B. The further revision task AB synchronizes the changes of its two predecessors A and B. This relation is represented by the successiveRT and previousRT links of A, B and AB. Furthermore the tracing of all item changes is accomplished easily because the repository has sufficient information for providing an extensive integration and consistency management. The documentRevision function explained in Definition 9 in section II can determine all information items of the model version of revision task AB. This resulting model version is depicted by Figure 8 and contains the expected changed model elements PrintMedia and author.

Thereby the application of the introduced merge technique on the concurrent evolved example model illustrates exemplarily an intuitive, optimistic merge of models in a collaborative development process.

IV. PROTOTYPE IMPLEMENTATION

The merge technique which was described in section II and III 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 III (e.g. changeState, mergeState etc.) were implemented as EMF-Operations and declarative specified as OCL-constraints. The EMF code generation creates a simple EMF-tree editor on basis of the repository specification. This editor allows modifying models which are stored in the repository. For the automatic integration of any meta-model, a development of an ATL transformation from the selected meta-model to an integrated repository scheme is planned. The OCL-specified functions (e.g. to calculate merges) can be called in the OCL-console provided by the Eclipse/OCL-plugin. The Model revisions can be calculated on request by the documentRevision-function via the OCL-console of the Eclipse platform. Concepts for identification of inconsistencies of involved model elements as described in [2] are also implemented in the prototype. Currently, this change-based repository concept for model development management will be implemented on basis of the integrated, Eclipse-based collaboration platform Jazz. Jazz already supports the collaborative management of source code...
but not of models.

V. CONCLUSIONS AND FURTHER WORK

This paper gives a proposal for a formal semantics-description of synchronizing documents and especially models which evolve concurrently. Therefore a three-way merge technique for synchronizing concurrent revisions of information item sets was introduced in section II. This clear describable and intuitive merge technique is not directly applicable to the well-known model representation formats but can be adapted for merging model structures independently of their meta-model. For this reason a special kind of model representation was introduced in section III. This representation defines a model representation module which can be managed as an information item by the introduced merge technique explained in section II. A consistency-preserving strategy is not a matter of this paper and is discussed in [2].

An implementation of the explained approach and its integration in a popular collaboration platform like Jazz is subject to further work.

ACKNOWLEDGMENT

My thanks to Tim Schumann for helpful discussions and suggestions.

REFERENCES

