Version Control in MDE

Adrian Rutle[1], Alessandro Rossini[2], Yngve Lamo[1] and Uwe Wolter[2]
[1]Faculty of Engineering, Bergen University College, Norway
[2]Department of Informatics, University of Bergen, Norway

ABSTRACT

In Model-Driven Engineering models are the primary artefacts of the software development process. Similar to other software artefacts, models undergo a complex evolution during their life cycles. Version control is one of the key techniques which enables developers to tackle this complexity. Traditional version control systems are based on the copy-modify-merge paradigm which is not fully exploited in MDE because of the lack of model-specific techniques. In this paper we give a formalisation of the copy-modify-merge paradigm in MDE. In particular, we analyse how common models and merge models can be defined by means of category-theoretical constructions. Moreover, we show how the properties of those constructions can be used to identify model differences and conflicting modifications.

1. INTRODUCTION AND MOTIVATION

Software models are abstract representations of software systems. These models are used to tackle the complexity of present-day software by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE) models are first-class entities of the software development process and undergo a complex evolution during their life-cycles. Therefore, the need for techniques and tools to support model evolution activities such as version control is increasingly growing. Present-day MDE tools offer a limited support for version control of models. Typically, the problem is addressed using a *lock-modify-unlock* paradigm, where a repository allows only one developer to work on an artefact at a time. This approach is workable if the developers know who is planning to do what at any given time and can communicate with each other quickly. However, if the development group becomes too large or too spread out, dealing with locking issues might become a hassle.

On the contrary, traditional version control systems such as Subversion enable efficient concurrent development of source code. These systems are based on the *copy-modify-merge* paradigm. In this approach each developer accesses a repository and creates a local working copy – a snapshot of the repository's files and directories. Then, the developers modify their local copies simultaneously and independently. Finally, the local modifications are merged into the repository. The version control system assists with the merging by detecting conflicting changes. When a conflict is detected, the system requires manual intervention of the developer.

Unfortunately, traditional version control systems are focused on the management of text-based files, such as source code. That is, difference calculation, conflict detection, and source code merge are based on a per-line textual comparison. Since the structure of models is graph-based rather than text- or tree-based [1], the existing techniques are not suitable for MDE.

Research has lead to various outcomes related to model evolution during the last years: [3] for the difference calculation, [2] for the difference representation and [4] for the conflict detection, to cite a few. However, the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as *add*, *delete*, *rename* and *move* are given different semantics in different works/tools. In addition, concepts such as *synchronisation*, *commit* and *merge* are only defined semiformally. Moreover, the terminology is not precise and unique, e.g. the terms "create", "add" and "insert" are frequently used to refer to the same operations.

Our claim is that the adoption of the copy-modify-merge paradigm is necessary to enable effective version control in MDE. This adoption requires formal techniques which are targeting graph-based structures. The goal of this paper is the formalisation of the copy-modify-merge paradigm in MDE. In particular, we show that common models and merge models can be defined as pullback and pushout, respectively. To achieve this, we use the Diagram Predicate Framework (DPF)[1] [5, 6, 7] which provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In addition, DPF enables us to define a language to represent model differences and a logic to detect conflicting modifications.

2. VERSION CONTROL IN MDE

First we start with an example to present a usual scenario of concurrent development in MDE. The example is obviously simplified and only the details which are relevant for our discussion are presented. Then, common models, merge models and their computations are analysed in the subsequent sections.

Suppose that two software developers, Alice and Bob, use a version control system based on the copy-modify-merge paradigm. The scenario is depicted in Fig. 1.

Alice checks out a local copy of the model V_1 from the repository and modifies it to V_{1A}, where 1 is a version number and A stands for Alice. This modification takes place in the *evolution step* e_{1A}. Since the model in the repository may have been updated in the mean time, she needs to synchronise her model with the repository in order to integrate

[1]Formerly named Diagrammatic Predicate Logic (DPL).

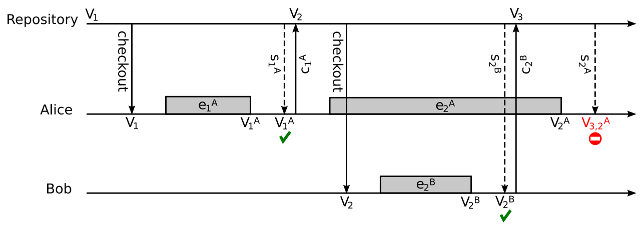

Figure 1: The scenario for the example

her local copy with other developers' modifications. This is done in the *synchronisation* s_{1^A}. However, no modifications of the model V_1 has taken place in the repository while Alice was working on it. Therefore, the synchronisation completes without changing the local copy V_{1^A}. Finally, Alice commits the local copy, which will be labelled V_2 in the repository. This is done in the *commit* c_{1^A}.

Afterwards, Bob checks out a local copy of the model V_2 from the same repository and modifies it to V_{2^B}. Then, he synchronises his model with the repository. Again, the synchronisation completes without changing the local copy V_{2^B}. Finally, Bob commits the local copy, which will be labelled V_3 in the repository.

Alice continues working on her local copy, which is still V_2. This model is not synchronised with the repository which contains Bob's modifications. She synchronises her model with the repository where the last model is V_3. Therefore, the synchronisation computes the *merge model* $V_{2^B,3}$. Now, the version control system may report a conflict in the merge model which forbids the commit c_{2^B}. The resolution of the conflict requires the manual intervention of Alice, who must review the model and decide to adapt it to Bob's modifications, or, adapt Bob's modifications to her own model.

2.1 Common Model

When Alice changes her local copy from V_2 to V_{2^A}, her development environment must keep track over what is common between the two models. The identification of what is common is the same as the identification of what is not modified, which should be feasible to implement in any tool.

Every two model elements which correspond to each other can be identified in a *common model.* For example, the model $C_{2^A,3}$ is the common model of the models V_{2^A} and V_3. The usage of a common model makes the construction of merge models at synchronisation step easier (explained in Sec. 2.2). In some frameworks, what is common between two models is defined implicitly by stating that structurally equivalent elements imply that the elements are equal (*soft-linking*). This approach has the benefit of being general, but its current implementations are too resource greedy to be used in production environment. In other frameworks, elements with equal identifiers are seen as equal elements (*hard-linking*). Unfortunately, this approach is tool-dependent, since the element identification is different for every environment. Our claim is that "recording" which elements are kept unmodified during an evolution step addresses the problems of the soft- and hard-linking approaches. That is, these equalities are specified explicitly in common models as in the following definition.

Definition 1. A model C_{i,i^U} together with the injective morphism inj_{V_i} and the inclusion morphism $inc_{V_{i^U}}$ is a common model for V_i and V_{i^U}.

$$\begin{tikzcd}
 & & C_{i,k} \arrow[dl, "inj_{C_{i,j}}"'] \arrow[dr, hook, "inc_{C_{j,k}}"] \arrow[ddll, bend right, "f"'] \arrow[ddrr, hook, bend left, "g"] & & \\
 & C_{i,j} \arrow[dl, "inj_{V_i}"'] \arrow[dr, hook, "inc_{V_j}"] & P.B. & C_{j,k} \arrow[dl, "inj_{V_j}"'] \arrow[dr, hook, "inc_{V_k}"] & \\
V_i & & V_j & & V_k
\end{tikzcd}$$

Figure 2: Common models: $C_{i,j}$ and $C_{j,k}$; and the composition: $C_{i,k}$

In order to find the common model between two models which are not subsequent versions of each other, i.e. for which we do not have a direct common model, we can construct the common model by the composition of the common models of their intermediate models. We call this common model for the *composition of commons* or the *normal form.* A possible way to compute this common model is as follows (see Fig. 2):

Proposition 1. Given the diagrams $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xhookrightarrow{inc_{V_j}} V_j$ and $V_j \xleftarrow{inj_{V_j}} C_{j,k} \xhookrightarrow{inc_{V_k}} V_k$, for $j = i{+}1$ and $k = j{+}1$, the common model for V_i and V_k is $C_{i,k}$ with the two morphisms f and g where $f = inj_{C_{i,j}}; inj_{V_i}$, $g = inc_{C_{j,k}}; inc_{V_k}$, and, $C_{i,k}$ is the pullback $(C_{i,k},\ inj_{C_{i,j}} : C_{i,k} \to C_{i,j},\ inc_{C_{j,k}} : C_{i,k} \to C_{j,k})$ of the diagram $C_{i,j} \xhookrightarrow{inc_{V_j}} V_j \xleftarrow{inj_{V_j}} C_{j,k}$ such that $inc_{C_{j,k}}$ is an inclusion.

2.2 Merge Model

Recall that when Alice wanted to commit her local copy V_{2^A} to the repository, she had to first synchronise it with the repository. In the synchronisation s_{2^A}, a merge model $V_{2^A,3}$ was created. The merge model must contain the information which is needed to distinguish which model elements come from which model. But this is exactly one of the properties of pushout; therefore, we use pushout construction to compute merge models, as stated in the next proposition. The properties of the pushout are then used to decorate merge models such that added, deleted, renamed and moved elements are distinguished (explained in Sec. 2.4).

Proposition 2. Given the models V_i, V_j and $C_{i,j}$, the merge model $V_{i,j}$ is the pushout $(V_{i,j},\ m_i : V_i \to V_{i,j},\ m_j : V_j \to V_{i,j})$ of the diagram $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xhookrightarrow{inc_{V_j}} V_j$ such that m_j is an inclusion.

2.3 Synchronisation and Commit

Fig. 3 outlines synchronisation and commit operations in the copy-modify-merge paradigm. These operations are defined as follows. In Fig. 3 and in the following definitions and propositions, U stands for a username.

Definition 2. Given the local copy V_{i^U}, the last model in the repository V_j and their merge model $V_{i^U,j}$, the synchronisation $s_{i^U} : (V_{i^U}, V_j) \to V_{j^U}$ is an operation which generates a synchronised local copy V_{j^U} such that

$$V_{j^U} := \begin{cases} V_{i^U} & \text{if } i = j; \\ V_{i^U,j} & \text{if } i < j, \text{ and } V_{i^U,j} \notin \mathcal{C}^U \end{cases}$$ where $\mathcal{C}^U$ is the set of conflicting merge models.

$$\begin{tikzcd}
& C_{i,i^U} \arrow[dl, hook', "inc_{V_{i^U}}"'] \arrow[dr, "inj_{V_i}"] \arrow[dd, phantom, "P.O."] & & C_{i,j} \arrow[dl, "inj_{V_i}"'] \arrow[dr, hook, "inc_{V_j}"] \arrow[dd, phantom, "P.O."] & \\
V_{i^U} \arrow[dr, hook', "mu_{i^U}"'] & & V_i \arrow[ll, Rightarrow, "e_{i^U}"'] \arrow[r, equal] \arrow[dl, "mu_i"'] \arrow[dr, "mr_i"] \arrow[dd, phantom, "P.O."] & \cdots \arrow[r, Rightarrow] & V_j \arrow[dl, hook', "mr_j"'] \arrow[d, Rightarrow, dashed] \\
& V_{i,i^U} \arrow[dr, "mu_{i^U,i}"'] & & V_{i,j} \arrow[dl, "mr_{i,j}"] & V_{j+1} \\
& & V_{i^U,j} \arrow[r, dashed] & V_{j^U} \arrow[ur, Rightarrow, "c_{i^U}"'] &
\end{tikzcd}$$

Figure 3: Synchronisation and Commit

Definition 3. Given the synchronisation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$, the commit $c_{i^U} : V_{j^U} \Rightarrow V_{j+1}$ is an operation which adds the model V_{j^U} to the repository as V_{j+1}.

Whenever a local copy V_{i^U} is synchronised with a model V_j from the repository, if the version numbers are the same, i.e. $i = j$, then a *synchronised* local copy V_{j^U} will be created such that $V_{j^U} = V_{i^U}$. However, if $i < j$, then a merge model $V_{i^U,j}$ will be created such that $V_{j^U} = V_{i^U,j}$ only if $V_{i^U,j}$ is not in a conflict state, i.e. $V_{i^U,j} \notin \mathcal{C}^U$. Finally, the commit operation will add the synchronised local copy V_{j^U} to the repository and will label it V_{j+1}. The next procedure explains the details of our approach to the synchronisation and commit operation.

Given the models V_{i^U}, V_i, C_{i,i^U} and V_j, where $i < j$, the synchronisation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ is computed as follows:

1. compute the merge model V_{i,i^U}
2. compute the common model $C_{i,j}$
3. compute the merge model $V_{i,j}$
4. compute the merge model $(V_{i^U,j},\ mu_{i^U,i},\ mr_{i,j})$ as the pushout of $V_{i,i^U} \xleftarrow{mu_i} V_i \xrightarrow{mr_i} V_{i,j}$
5. $V_{j^U} := V_{i^U,j}$ only if $V_{i^U,j} \notin \mathcal{C}^U$

2.4 Difference and Conflict

As mentioned, during a synchronisation operation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ where $i < j$, the merge model $V_{i^U,j}$ may contain conflicts. To detect these conflicts, we need a way to identify the differences between V_{i^U} and V_j, i.e. the modifications which has occurred in the evolution step(s). Difference identification in the merge model $V_{i^U,j}$ can be done by distinguishing common elements, V_{i^U}-elements and V_j-elements from each other. However, since this is one of the properties of merge models, we only need a language to express the differences.

Since models are graph-based, we need a diagrammatic language to tag the model elements as *common*, *added*, *deleted*, *renamed* and *moved*. Therefore, we use DPF to define such a diagrammatic language. In DPF each modelling language L corresponds to a diagrammatic signature Σ_L and a metamodel MM_L. L-models are represented by Σ_L-specifications which consist of a graph and a set of constraints. The graph represents the structure of the model, and predicates from Σ_L are used to add the constraints to the graph [7].

We define a signature $\Sigma_\triangle$ for our language. $\Sigma_\triangle$ consists of five predicates: `[common]`, `[add]`, `[delete]`, `[rename]`, and `[move]`. The merge models will be decorated by predicates from the signature $\Sigma_\triangle$ in addition to the predicates from Σ_L. For example, an element in V_{i,i^U} will be tagged with `[add]` if it does not exist in V_i, with `[delete]` if it does not exist in V_{i^U}, and with `[common]` if it exists in both.

We have also developed a logic for $\Sigma_\triangle$ which is used for two main purposes: to obtain the synchronised local copy V_{j^U} from $V_{i^U,j}$; and to detect conflicts in the decorated merge model $V_{i^U,j}$. The synchronised local copy is obtained by interpreting the predicates as operations, e.g. if an element is tagged with the predicate `[delete]`, it will not exist in V_{j^U}. The detection of conflicts is done by checking the tagged elements against some predefined set of conflicting modifications, e.g. if an element is tagged with `[rename]` twice, it is in conflict. Although conflicts are context-dependent, we have recognised some situations where syntactic conflicts will arise. The following is a summary of the concurrent modifications which we identify as conflicts:

– adding structure to an element which has been deleted
– renaming an element which has been renamed
– moving an element which has been moved

3. SUMMARY AND FUTURE WORK

The copy-modify-merge paradigm is proven to be an effective solution to tackle the complexity of version control of text-based artefacts. In this paper, we have shown how this paradigm can be exploited for version control of graph-based structures such as software models. We have formalised the concepts of the copy-modify-merge paradigm in MDE, by means of pullback and pushout constructions. In addition, we have defined a proof-of-concept diagrammatic language for the representation of model differences, and a logic for the detection of syntactic conflicts.

In a future work, we will consider semantic conflicts. Moreover, a prototype implementation will be developed to verify the efficiency of the proposed techniques.

4. REFERENCES

[1] L. Baresi and R. Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *ICGT 2004*, volume 3256 of *LNCS*, pages 431–433. Springer, 2004.

[2] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *JOT*, 6(9):165–185, October 2007.

[3] EMF Compare. *Project Web Site.* `http://www.eclipse.org/emft/projects/compare/`.

[4] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *ENTCS*, 127(3):113–128, 2005.

[5] A. Rutle, U. Wolter, and Y. Lamo. Diagrammatic Software Specifications. In *NWPT 2006*, October 2006.

[6] A. Rutle, U. Wolter, and Y. Lamo. A Diagrammatic Approach to Model Transformations. In *EATIS 2008*, pages 1–8. ACM, 2008.

[7] A. Rutle, U. Wolter, and Y. Lamo. A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland, 2008.