Collaborative Editing of EMF/Ecore Meta-models and Models
Conflict Detection, Reconciliation, and Merging in DiCoMEF

Amanuel Koshima and Vincent Englebert
PReCISE Research Center, University of Namur, Namur, Belgium

Keywords: EMF, DSML, Collaborative Modeling, Conflict Detection, Merging.

Abstract: Despite the fact that Domain Specific Modeling tools become very powerful and more frequently used, the support for their cooperation has not reached its full strength and demand for model management is growing. In cooperative work, the decision agents are semi-autonomous and therefore a solution for reconciliating DSM after a concurrent evolution is needed. Conflict detection and reconciliation are important steps for merging of concurrently evolved (meta)models in order to ensure collaboration. In this work, we present a conflict detection, reconciliation and merging framework for concurrently evolved meta-models and models. Besides, we formally specify the EMF Ecore meta-model into set constructs that help to analyze the (meta)model and operations performed on it.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Domain Specific Modeling languages (DSML) have matured and used as an efficient alternative to General Purpose Modeling languages (e.g., UML, Petri Nets) for modeling complex systems (Kelly, 1998). DSML defines the structure, behavior and requirements of software applications in specific domain by using domain concepts rather than generic modeling languages (Schmidt, 2006). The benefits of using DSML have been described in (Kelly and Tolvanen, 2008). DSML describes concepts at different levels of abstraction using models, meta-models and meta-meta-models. A model is an abstraction of a software system and a meta-model is a DSL oriented towards the representation of software development methodologies and endeavors (Gonzalez-Perez and Henderson-Sellers, 2008). A meta-meta-model is a minimum set of concepts which defines languages (i.e. MOF (Object Management Group (OMG), 2002), MetaL (Englebert and Heymans, 2007), EMF/Ecore (Steinberg et al., 2009)). A meta-meta-model specifies all the concepts and constraints that are used and respected, respectively, by the meta-model (a meta-model conforms to a meta-meta-model). Besides, a meta-meta-model describes itself.

Since 90's several metaCASE tools have been developed such as Atom3 (de Lara and Vangheluwe, 2002), GME (Ledeczi et al., 2001), MetaDone (Englebert and Heymans, 2007), or MetaEdit+ (Kelly, 1998). These tools give an ad-hoc environment that enables method engineers to edit and manage models and/or meta-models. However, most of these metaCASE tools consider the modeling process as a single user task even though modeling of software systems usually requires collaboration among members of a group with different scopes and skills (i.e. middleware engineers, human interface designers, database experts, business analysts) (Koshima et al., 2011; Koshima et al., 2013). Therefore, there is a need for metaCASE tools to support sharing of modeling artifacts (i.e., model and meta-models) and managing and synchronization of activities of the group members.

In collaborative modeling, different members of a group could concurrently edit shared modeling artifacts throughout the development life cycle of a software application. As a result, the shared modeling artifacts might not seamlessly work together or the final result may not be what users want. In other words, these modeling artifacts become inconsistent with each other. The main challenge of collaborative modeling is to detect inconsistencies and conflicts and to resolve them. These conflicts could be *textual, syntactic, or semantic* (Mens, 2002). Although there are some approaches for detecting conflict in text or tree based documents, these approaches are not suitable for models that have a graph based nature (Altmanninger et al., 2009; Mougenot et al., 2009). These approaches might neglect the syntax and semantics of models. Our work focuses on syntactic (structural)

Koshima A. and Englebert V..
Collaborative Editing of EMF/Ecore Meta-models and Models - Conflict Detection, Reconciliation, and Merging in DiCoMEF.
DOI: 10.5220/0004709500550066
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 55-66
ISBN: 978-989-758-007-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

conflicts. Semantics conflicts are difficult to solve and are considered in our approach as an extra layer on top of the proposed framework.

The commonly adopted approach to ensure collaboration is a central repository with merge mechanisms and locking techniques (Mougenot et al., 2009). However, the locking technique is not scalable for a large number of users who work in parallel (Altmanninger et al., 2009; Mens, 2002) and it takes much time to resolve conflicts in practice (Altmanninger et al., 2009; Pilato et al., 2008). This approach also restricts users to be dependent on one repository and it may introduce unnecessary access right bureaucracies that lead to dissatisfaction among members of a group. For instance, MetaEdit+ (Kelly, 1998) implements *Smart Mode Access Restricting Technology* (Smart Locks ©) to support concurrent access of shared modeling artifacts that are stored centrally. EMFStore (Koegel and Helming, 2010) uses a central repository with copy-merge techniques to ensure collaboration.

There is another mode of collaboration that consists of a group of people concerned by a cooperative task that is large, transient, not stable or even non deterministic (Schmidt and Bannon, 1992). The interaction among members of a group could be dynamic and users are semi-autonomous in their partial work. Each member has his/her own copy of a shared modeling artifact and carries on his/her activity in isolation with other users or a central authority. Users communicate their work by sending messages to other members (Mougenot et al., 2009). This mode of collaboration gives users a better control over their data and it mitigates being dependent on single repository (modeling artifacts are distributed among members of a group). But, it is challenging to keep all copies of modeling artifacts consistent; because they could be modified concurrently by users.

D-Praxis (Mougenot et al., 2009) is a peer-to-peer collaborative model editing framework that relies on Lamport clock and delete semantics to automatically solve conflicts. This framework has a “lost-update” problem and we argue that final results of automatic reconciliation process could not reflect the intention of users. In this paper, we present a distributed collaborative model editing framework called DiCoMEF that ensures collaboration among DSM tools (Koshima et al., 2011; Koshima et al., 2013). DiCoMEF lets each member of a group to have his/her own local copy of shared modeling artifacts. In DiCoMEF, modifications are controlled by human agents (not automatic).

This paper extends our previous work (Koshima et al., 2011; Koshima et al., 2013) by giving the formal specification of (meta)models using set constructs. It also provides a detailed description of conflict detection and reconciliation processes. The paper is organized as follows: Section 2 describes DiCoMEF framework. Section 3 gives a formalization of EMF/Ecore (meta)model. In Section 4, the history meta-model of DiCoMEF is described in section 4.1. Section 4.3 presents the conflict detection strategy and section 4.2 specifies the reconciliation framework. Finally, section 5 describes the future work and conclusion.

2 DiCoMEF

DiCoMEF (Koshima et al., 2011; Koshima et al., 2013) is a distributed collaborative model editing framework for EMF (meta)models where each member of a group has his/her own local copy of a (meta)model. The main concepts used in DiCoMEF are *person, role, role type, model, meta-model, copy model and master model*. A master (meta)model is the main (meta)model which has one or more copy (meta)models that are distributed among editors and observers. DiCoMEF uses a universal unique identifier (UUID) to differentiate (meta)model elements (i.e. classes, attributes, references) uniquely. Two (meta)model elements are considered as identical if and only if they have the same UUID. Besides, a person involved in collaborative modeling has a role, which is typed as a *controller, editor* or *observer*. In fact, there are two controller role types which are implemented in DiCoMEF such as a model controller or a meta-model controller.

Model (resp. meta-model) controllers are software configuration managers who manage evolutions of a master (resp. meta-) model. A controller role type is flexible meaning that it can be assigned (delegated) to other members of a group as long as there is one unique coordinator per group. A person who has an editor role can write and read his/her local copy (meta)models, whereas an observer role only gives a read access to a local copy (meta)models.

DiCoMEF relies on two concepts such as *Main-line* and *branches* in order to store models and meta-models. Besides, it uses these two concepts to facilitate communications among members of a group (see Fig. 1)[1]. The main-line stores different versions of a copy (meta)model locally at each editors site. An editor does not have a write access to modify a copy (meta)model stored on the main-line. Rather s/he fist creates a branch from the main-line and modifies the

[1]Although these terms are also used by SCM programs, our framework does not rely on a central SCM.

(meta)model there. In order to communicate local modifications with other members, s/he sends her/his local modifications to a controller as a change request. The controller propagates accepted changes to all members of the group and changes propagated from the controller are applied on the main-line. For example, Fig. 1 shows an evolution of a copy (meta)model from version V_0 to version V_1 on the main-line based on changes propagated from a controller. Besides, it indicates a local modification performed by an editor on the branch that evolves a copy (meta)model from version V_0 to version $V_{0.1}$; a branch was created before a copy (meta)model evolves from version V_0 to version V_1.

The communication framework of DiCoMEF is organized around the controller that acts as a central hub wrt. his/her (meta)model he/she is responsible. This could be a limitation of DiCoMEF, but at the same time it might be considered as its strength as well. Indeed, DiCoMEF provides a technical framework over which different communication strategies can be employed using method engineering techniques (e.g., delegation mechanisms, pooling). For example, a token can be used and whoever has a token is a controller who can modify a (meta)model and propagates changes.

In DiCoMEF, when members of a group modify (meta)models locally, elementary change operations (*create, delete and updates*) are stored locally in a local repository. These elementary operations constitute a history that is used to propagate local modifications to the controller and secondarily to other members. Histories are defined by a history meta-model. The history meta-model, conflict detection, and reconciliation are discussed later. A detailed description of DiCoMEF is found in (Koshima et al., 2011; Koshima et al., 2013).

A *change request* is a set of local modifications that are performed by an editor and sent to a controller in order to share local modifications with other members (commit changes). A change request could be either accepted, rejected, or modified by a controller before being committed to the main-line (and then shared with other members). A controller works by consulting a rationale of modification or an editor who proposed the change request in case of conflicts. Afterwards, if the change request is accepted, the controller sends a *change propagation* to all members so as to evolve (meta)models. (Meta)models on the main-lines evolve automatically, whereas (meta)models on branch evolve when a user updates a branch based on the change propagation. For example, in Fig. 1, a copy (meta)model evolves from version V_0 to version V_1 on the main-line based on changes propagated from a controller. It also shows a branch that is created by an editor to modify a copy (meta)model locally from version V_0 to version $V_{0.1}$; a branch was created before a copy (meta)model evolves from version V_0 to version V_1.

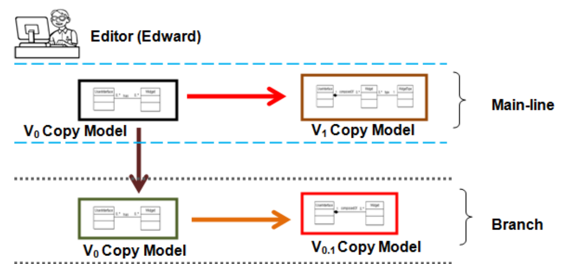

Figure 1: Main-line and Branch.

We have implemented DiCoMEF as an Eclipse plug-in that captures the history of (meta)model adaptation when the user edits tree views or uses the GMF editor (Graphical Modeling Framework (GMF), 2013). The communication framework of DiCoMEF is implemented using Java Messaging Service (Monson-Haefel and Chappell, 2000): users can exchange modifications via email. DiCoMEF has a MessageListnerImp that implements an IMessageListner interface and checks whether there is a new email message or not. If it receives a new email message, it downloads the file and updates the DiCoMEF repository automatically.

3 FORMALIZATION OF MODELS

Some research work has been done in the past to formally specify an EMF meta-model using graph theory (Taentzer et al., 2012). In (Monperrus et al., 2009), the authors used a set theory to define an abstraction level of MOF (Object Management Group (OMG), 2002). This work used a set theory to formalize an EMF Ecore model (Steinberg et al., 2009) because we believe that most people are familiar with the set theory as a result it is easy for people to understand and reason about models.

3.1 Notation

In this paper, we will use several ad-hoc notations that are defined in this preliminary section. In a binary cartesian products, the identifying component is underlined. If $R \subseteq \underline{A} \times B$ then $\forall a \in A, \forall b_1, b_2 \in B : (a,b_1),(a,b_2) \in R \implies b_1 = b_2$ and $R(a) = b \triangleq (a,b) \in R$. The presence of partial orders can be indicated with the $<$ superscript: $R \subseteq A \times B^<$ means

that tuples with a common element in first position are ordered $(a,b_1) < (a,b_2) < (a,b_3)$ wrt a and this information can be abbreviated as $R(a) = [b_1,b_2,b_3]$. The position (index) of an element in the list is represented with *pos* function as follows $pos(b_2,R(a)) = 1$ and $[e_1,\dots,e_2] - i \triangleq [e_1,\dots,e_{i-1},e_{i+1},\dots,e_n]$. If R is a binary relation $\subseteq A \times B$, then we note R^- the inverse relation $\subseteq B \times A$: $(a,b) \in R \Leftrightarrow (b,a) \in R^-$ and we note $R^\star$ its transitive closure, i.e. $\{(a,b) \dagger (a,b) \in R \vee \exists c : (a,c) \in R \wedge (c,b) \in R^\star\}$. 2^S denotes the powerset of a set S and $A \mapsto B$ a mapping function from set A to set B.

3.2 The Ecore Meta-meta-Model

Eclipse Modeling Framework(EMF) is widely used to build tools and applications. It generates codes (i.e. classes for the meta-model and adapter classes for viewing and editing models) based on the structured data model (Steinberg et al., 2009). A model can be expressed using annotated Java interfaces, XML Schema, or UML modeling tools. EMF provides a facility to generate one form of representation from the other (using the EMF framework). EMF uses Ecore as a meta-meta-model to define different DSL languages and itself. Figure 2 shows the UML class diagram of the Ecore meta-model. The association depicted with blue color are derived associations where as the black lines are non-derived associations.

The root element of an Ecore meta-meta-model is an EPackage. An EPackage contains zero or more sub-packages and EClassifiers (i.e. EClass, EDataType, EEnum). A model class is represented by using an EClass, which is identified by a name and has zero or more attributes and references. A class can have zero or more super types. It can have zero or more operations. Properties (attributes) of a class are modelled using an EAttribute, which has a name and a type. Associations are modelled by EReference(s). An EReference models an end of an association between two classes; it has a name and a type (the EClass at the opposite end of the association). A bi-directional navigable association is modelled using two references that are related to each other by an eOpposite link. Besides, a composition association is represented by setting a containment boolean property of an EReference to `true`. The cardinality of a reference is modeled by setting lowerBound and upperBound values. Like references, an attribute's cardinality could be specified using lowerBound and upperBound features. The Ecore meta-meta-model is attached in the appendix and we also invite interested readers to refer to (Steinberg et al., 2009).

[2]http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/doc-files/EcoreRelations.gif

3.2.1 Semantics

The semantics of the Ecore meta-meta-model is formally defined by a systematic mapping of its structural elements onto mathematical constructs.

We define a set Σ that encompasses a set of constraints expressed in some language that is not relevant here. For each class C in Ecore, we define a set E_C. For each association r between classes A and B in Ecore, we define a set $\rho_r \subseteq E_A \times E_B$. Let's observe that in all the Ecore meta-meta-model diagrams published so far, the relations denote accessor methods and not sets of tuples as specified in the UML standard (UML 2.0 superstructure, 2011). For this reason, multiplicities in our mapping may not match the cardinality of the accessor links in the diagrams published so far. The product denoting this association is annotated with the $\underline{\dots}$ and $^<$ symbols depending on its semantics in Ecore: *is the association ordered? is it one-to-many, or many-to-many?* For each attribute a of type T in class C, a set $\alpha_a \subseteq E_C \times T$ is defined where $T \in E_D$.

Inheritance between classes is mapped to inclusion constraints between the corresponding sets, hence, if A isa B, then the constraint $E_A \subseteq E_B$ is added to Σ. When the superclass is abstract, the inclusion is replaced with the equality operator. We bootstrap first the process by defining some sets:

$$\begin{aligned} E_D &= \{EString, EInteger, \dots\} \\ EString &= \{`', `a', `aa', `ab', \dots\} \\ EInteger &= \{EInteger.min, \dots, -1, 0, 1, \dots, EInteger.max\} \\ &\dots \end{aligned}$$

E_D elements are data types. In EMF, a data type denote simple data types in Java, classes, interfaces, and arrays that are not modeled using with E_C elements (Steinberg et al., 2009). We define Val as the union of all data type values: $Val = \cup_{T \in E_D} T$.

Ecore classes in the meta-meta-model are mapped to sets: E_C (aka EClass), E_D (aka EDataType), E_P (aka EPackage), E_R (aka EReference), E_A (aka EAttribute), E_E (aka EEnum), E_L (aka EEnumLiteral), E_O (aka EOperation), E_{PA} (aka EParameter), E_{AN} (aka EAnnotation), E_{ne} (aka ENamedElements), E_{me} (aka EModelElement), E_c (aka EClassifier), E_{te} (aka ETypedElement), E_{sf} (aka EStructuralFeature) and E_{OB} (aka EObject). Lower case subscripts denote abstract classes. For the sake of simplicity, EFactory and EStringToStringMapEntry are not considered in this work.

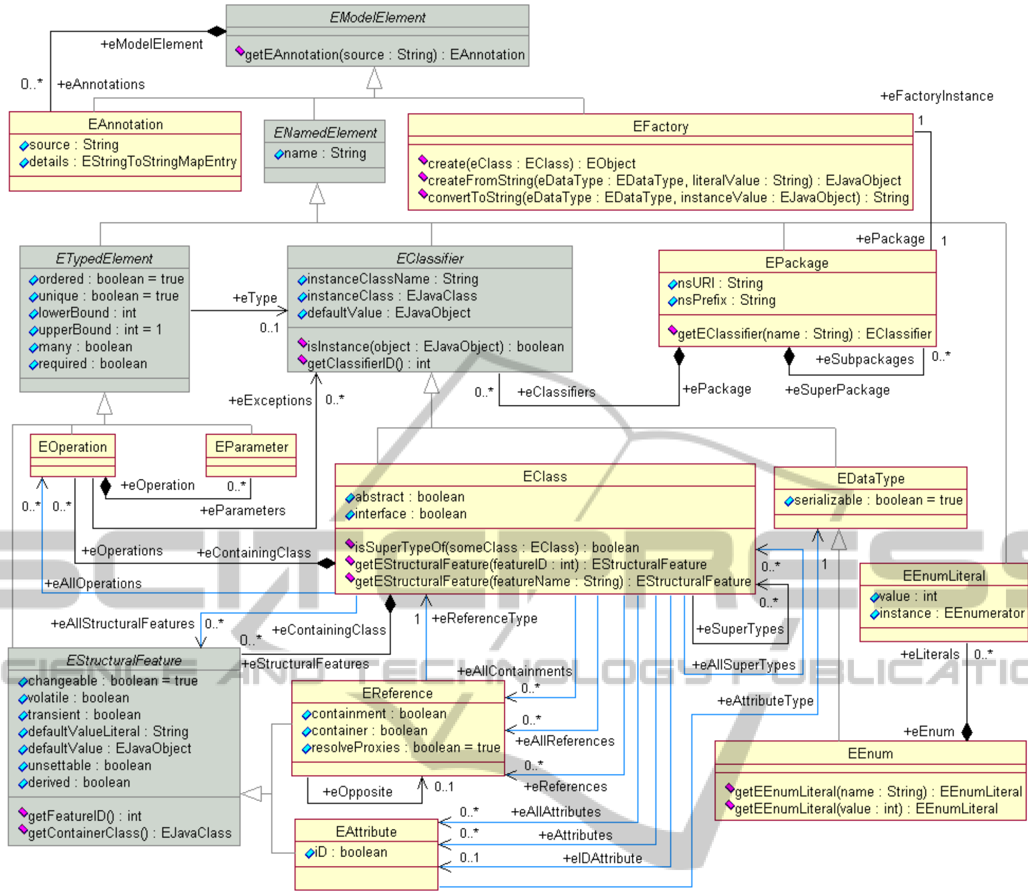

Figure 2: Ecore Meta-model [2].

The inheritance relationship between non-abstract class and its subtypes are modeled using set inclusion constraints and the quality if the supertype is abstract. The base class of all Ecore model elements is an EObject.

$$E_{me} = E_{OB} \qquad E_{AN} \cup E_{ne} = E_{me}$$
$$E_{te} \cup E_c \cup E_L \cup E_P = E_{ne} \qquad E_O \cup E_{PA} \cup E_{sf} = E_{te}$$
$$E_C \cup E_D = E_c \qquad E_A \cup E_R = E_{sf}$$
$$E_E \subseteq E_D$$

In this formalization, we don't consider associations that denote derived associations or facilities to access objects neither opposite associations. Each relevant association is translated as a relation between its ends.

$\rho_{eClassifiers} \subseteq E_P \times E_c{}^{<}$

$\rho_{eSubPackages} \subseteq E_P \times E_P{}^{<}$

$\rho_{eSuperTypes} \subseteq E_C \times E_C{}^{<}$

$\rho_{eSF} \subseteq E_C \times E_{sf}{}^{<}$ (*eSF* is a shortcut for *eStructuralFeatures*)

$\rho_{eIDAttribute} \subseteq \underline{E_C} \times E_A$

$\rho_{eType} \subseteq \underline{E_{te}} \times E_c$

$\rho_{eOpposite} \subseteq \underline{E_R} \times E_R$

$\rho_{eKeys} \subseteq E_R \times E_A{}^{<}$

$\rho_{eOperations} \subseteq E_C \times E_O{}^{<}$

$\rho_{eParameters} \subseteq E_O \times [E_{PA}]^{<}$

$\rho_{eExceptions} \subseteq E_O \times E_c$

$\rho_{eLiterals} \subseteq E_E \times E_L{}^{<}$

$\rho_{eAnnotations} \subseteq E_{me} \times E_{AN}{}^{<}$

$\rho_{details} \subseteq E_{AN} \times E_M{}^{<}$

For sake of simplicity, we define $\mathsf{owner}(sf) \triangleq \rho^{-}_{eSF}(sf)$ and $\mathsf{type}(te) \triangleq \rho_{eType}(te)$ as respectively the owner of a structural feature and its type. Σ is

completed with all the integrity constraints defined for Ecore such as “EPackage must have unique names” or “the values of the lowerbound attribute must less or equal than the value of the upperbound attribute for a same class”, ...

An Ecore meta-model MM is thus defined as a tuple of sets:

$$(E_C, E_A, \ldots, \rho_{eClassifiers}, \rho_{eSubPackages}, \ldots, \alpha_{name}, \ldots, \Sigma)$$

Example: A simple Petri net meta-model (Fig. 3 depicts its class diagram) could be defined as:

$$\begin{aligned}
E_C &= \{Pl, Tr, Nt, Ne\} \\
E_A &= \{Ne.name, Pl.tokens\} \\
E_R &= \{Pl.to, Pl.from, Tr.from, Tr.to, \\
&\quad Nt.places, Nt.transitions\} \\
E_P &= \{PN\} \\
\rho_{eSF}(Ne) &= [Ne.name] \\
\rho_{eSF}(Pl) &= [Pl.tokens, Pl.to, Pl.from] \\
\rho_{eSF}(Tr) &= [Tr.name, Tr.to, Tr.from] \\
\rho_{eSF}(Nt) &= [Nt.places, Nt.transitions] \\
\alpha_{eClass.name} &= \{(Pl, \text{‘}Place\text{’}), (Tr, \text{‘}Transiton\text{’}), \\
&\quad (Nt, \text{‘}Net\text{’}), (Ne, \text{‘}NamedElement\text{’}), \\
&\quad (Ne.name, \text{‘}name\text{’}), (Pl.tokens, \text{‘}tokens\text{’}), \\
&\quad (Pl.to, \text{‘}to\text{’}), (Pl.from, \text{‘}from\text{’}), \\
&\quad (Tr.to, \text{‘}to\text{’}), (Tr.from, \text{‘}from\text{’}), \\
&\quad (Nt.places, \text{‘}places\text{’}), \\
&\quad (Nt.transitions, \text{‘}transitions\text{’})\} \\
&\ldots
\end{aligned}$$

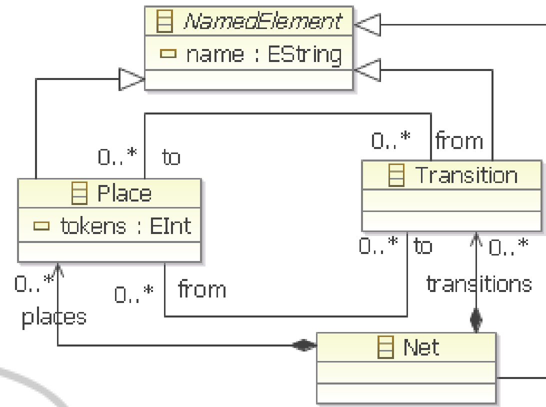

Figure 3: Petri net meta-model.

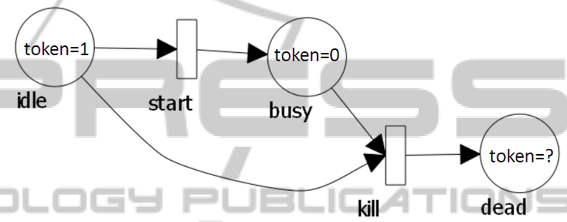

Figure 4: Petri net instance model.

3.3 Instantiation

If MM is a meta-model, a model M compliant with MM (noted M/MM) is defined as a tuple $([\![.]\!]_C, [\![.]\!]_A, [\![.]\!]_R, \Sigma_M)$ where each component is defined hereafter (E_{OB} denotes an infinite set of objects):

- $[\![.]\!]_c : E_C \mapsto 2^{E_{OB}}$ A class is modeled as a set of objects.
- $[\![.]\!]_a : E_A \mapsto (E_{OB} \times T^<)$ where T is the type of the attribute ($T \in E_D$). An attribute associates an object with values of type T.
- $[\![.]\!]_r : E_R \mapsto (E_{OB} \times E_{OB}{}^<)$ Same for references, but with objects.

We define $\tau(o) : E_{OB} \mapsto E_C$ the function that maps an object o to its class $c \in E_C$ such that $o \in [\![c]\!]_c \wedge \neg\exists d \in E_C : \rho_{eSuperTypes}(d, c) \wedge o \in [\![d]\!]_c$.

Example: A petri net instance model (depicted as an object diagram in Fig. 4) can be formalized as:

$$\begin{aligned}
&[\![Nt]\!]_c = \{net\} \\
&[\![Pl]\!]_c = \{idle, busy, dead\} \\
&[\![Tr]\!]_c = \{start, kill\} \\
&[\![Pl.name]\!]_a = \{(idle, \text{‘}idle\text{’}), (busy, \text{‘}busy\text{’}), \\
&\qquad (dead, \text{‘}deadlock\text{’})\} \\
&[\![Tr.name]\!]_a = \{(start, \text{‘}start\text{’}), (kill, \text{‘}kill\text{’})\} \\
&[\![Pl.tokens]\!]_a = \{(idle, 1), (busy, 0)\} \\
&[\![Pl.to]\!]_r(idle) = [start, kill] \\
&[\![Pl.from]\!]_r = \{(dead, kill), (busy, start)\} \\
&[\![Tr.to]\!]_r = \{(start, busy), (kill, dead)\} \\
&[\![Tr.from]\!]_r = \{(start, idle), (kill, idle)\} \\
&[\![Nt.place]\!]_r(net) = [idle, busy, dead] \\
&[\![Nt.transition]\!]_r(Nt.transition) = [start, kill]
\end{aligned}$$

As explained in Section 3.1, we used the notation $R(a) = [b_1, b_2, b_3]$ to resume $(a, b_1) < (a, b_2) < (a, b_3)$ for the $[\![]\!]_r$ construct.

3.4 Reflexivity

Since the base class of all Ecore model elements is EObject, this implies that the Ecore meta-meta-model may specify itself (reflexive definition): the Ecore meta-meta-model can then be modeled as an Ecore

meta-model (e.g. MM_{Ecore}). We could expect to observe the same property in our semantics framework of Ecore. Because of space constraints, we only present an partial definition of MM_{Ecore}:

$$
\begin{aligned}
E_C &= \{E_{ne}, E_P, E_c, E_R, \dots\} \\
E_A &= \{E_{ne}.name, E_P.nsURI, E_P.nsPrefix, \\
&\quad E_c.instanceClassName, E_c.instanceTypeName\} \\
E_R &= \{E_P.eSubPackages, E_P.eClassifiers, \\
&\quad E_c.eStructuralFeatures\} \\
E_P &= \{Ecore\} \\
\rho_{eSF} &= \{(E_{ne}, (E_{ne}.name)), \\
&\quad (E_P, (E_P.nsURI, E_P.nsPrefix, \\
&\quad E_P.eClassifiers, E_P.eSubPackages)), \\
&\quad (E_c, (E_c.instanceClassName, \\
&\quad E_R.containment, E_c.eStructuralFeatures))\}
\end{aligned}
$$

$$
\begin{aligned}
\alpha_{ne.name} = \{&(E_{ne}, \text{`}ENamedElement\text{'}), \\
&(E_P, \text{`}EPackage\text{'}), (E_c, \text{`}EClassifier\text{'}), \\
&(E_{ne}.name, \text{`}name\text{'}), (E_P.nsURI, \text{`}nsURI\text{'}), \\
&(E_P.nsPrefix, \text{`}nsPrefix\text{'}), \\
&(E_c.instanceClassName, \text{`}instanceClassName\text{'}), \\
&(E_c.instanceTypeName, \text{`}instanceTypeName\text{'})\} \\
&\dots
\end{aligned}
$$

$$
\begin{aligned}
\rho_{eType}(E_{ne}.name) &= EString \\
\rho_{eType}(E_c.instanceClassName) &= EString \\
\rho_{eType}(E_c.instanceTypeName) &= EString \\
\rho_{eType}(E_c.eStructuralFeatures) &= E_{sf}
\end{aligned}
$$

$$
\begin{aligned}
\rho_{eType}(E_P.nsPrefix) &= EString \\
\rho_{eType}(E_P.nsURI) &= EString \\
\rho_{eType}(E_P.nsURI) &= EString \\
\rho_{eType}(E_P.eSubPackages) &= E_P \\
\rho_{eType}(E_P.eClassifiers) &= E_c \\
\rho_{eSuperTypes}(E_c) &= E_{ne} \\
\rho_{eSuperTypes}(E_P) &= E_{ne} \\
\rho_{eClassifiers}(E_P) &= E_c
\end{aligned}
$$

This process could be continued with the other constructs of the Ecore meta-model and it shows that we can seamlessly define an Ecore meta-model by using itself meaning that our formalization support the reflexive nature of Ecore. Moreover, the constructs are used to define the semantics of a meta-model *MM* at the meta-model level or at the model level when it is reified and consistent. This last point is not discussed in this paper due to lack of space.

4 THE COLLABORATIVE MODEL

In collaborative modeling as it was mentioned in Section 1, (meta)models are concurrently edited by different members of a group. Later, these concurrently edited (meta)models need to be integrated (merged), but most of the time they might not seamlessly work together as a result of inconsistent modifications (conflict). These conflicts should be identified and resolved.

DiCoMEF uses a human controller to manage the evolution of (meta)models. S/He is assumed to be a business domain expert and who has a good modeling experience. Besides, s/he has a right to accept or reject change requests received from users. Once a new release is available, changes are propagated to all users who must take them into consideration before their own operations. In DiCoMEF, the controller role can be assigned or delegated to other members. This could help to facilitate collaboration among users with different expertise (database, user interface design, business domain, ...).

In case of conflict with his/her local change, DiCoMEF supports a semi-automatic conflict reconciliation strategy. Later, a user can send his/her local modifications merged with the last release of the model as a change request to the model controller. DiCoMEF provides users a facility to compose changes so as to put them in a same context (i.e., refactoring changes). This could later help user to understand changes during reconciliation process. It also lets users to annotate rationale of changes with multimedia files (i.e., audio, video, image, or text). During the reconciliation process, users can consult them to better understand rationale of changes and resolve conflicts.

4.1 Definition of History Meta-model

Change operations are used to exchange (meta)models modifications between users (Blanc et al., 2009). Besides, they are also used to detect conflicts and help the reconciliation process. Hence, it is important to specify the change operations unambiguously and formally.

A history meta-model has been defined to capture the information denoted by the change operations (*create, delete and updates*[3]) of models: a model element can be created (or deleted), a value of a single valued attribute or reference might be set. Besides,

[3] Let's note that read operations are not taken into consideration.

a new value can be added (or removed) to a multi-valued attribute or reference. Once this information is captured locally by this meta-model, an history can later be exchanged with other members.

Some works in the past have already used history meta-models. Hismo (Demeyer et al., 2001; Gîrba et al., 2005), a history meta-model based on a FAMIX meta-model, is an artificial modification history. Indeed, it transforms a snapshot model like UML or FAMIX into a history rather than recording modifications whenever they occur. It lacks preserving their exact time sequences. EDAPT (Herrmannsdoerfer, 2009), previously called COPE, is a tool based on EMF/Ecore meta-model that captures edit operations of meta-model adaptations whenever they occur. EMFStore (Koegel and Helming, 2010) uses a history meta-model to capture adaptation of instance models but does not work well with meta-model adaptations.

Since the Ecore meta-model is reflexive, constructs used to define the Ecore meta-model can be reused to define an Ecore model and its instances. The same history meta-model can thus capture both meta-model and model adaptations seamlessly. By the time this research was conducted, the history meta-model of EMFStore was tightly coupled with other components of the EMFStore implementation. As a result, EMFStore cannot be used/installed as an autonomous component for capturing history of meta-model adaptation [4]. Hence, we have extended EDAPT to capture both the adaptations of model and meta-model as part of the DiCoMEF implementation.

The history meta-model should fulfil the following requirements in order to be efficiently used in distributed collaborative (meta)model editing framework.

(R1) Self Contained: it must not have links (references) to model elements (surrogate technique should be used to reference model elements).

(R2) Universal Unique Identifier (UUI): it should have unique identifiers that identify change operations (create, set, delete, ...). Besides, it should also have UUIs for identifying (meta)model elements uniquely.

(R3) Composition: it has allow users to create composite of changes from other changes or composite changes.

(R4)Meta-model Adaptation: it has to capture meta-model adaptation operations.

(R5) Model Adaptation: it has to capture model adaptation operations.

[4]Recently, EMFStore has had refactoring to reduce a coupling between parts of the implementation that captures history with rest of implementation.

Table 1: Comparison of EMFStore, EDAPT, DiCoMEF.

	R1	R2	R3	R4	R5	R6	R7	R8	R9
EDAPT				√		√		√	
EMFStore	√	√	√		√				√
DiCoMEF	√	√	√	√	√	√	√	√	√

(R6) Understandability: users intention must be easy to understand. For example, EDAPT represents a changing of a parent element of a model element with a Move operation, which is easier to understand than EMFStore, which models the same modification with a composite operation that is composed of remove and add operations.

(R7) Multimedia Annotation: it has to give a user with a facility to annotate his rationale with multimedia files.

(R8) Cascade Operations: a delete operation should capture cascade operations that are caused by it. For example, when an EClass is deleted from an EPackage, all the references that point to the deleted EClass should be set to null. The delete operation should contain reference operations (copy of them) that set null value (or remove the deleted EClass from a collection). This could help only to roll back conflicting operations during merging process (to reconstruct references that are set to null or deleted due to the deletion of a model element). Roll back is different from undo operation that store operations in the stack. Roll back could be applied when an editor is closed and re-opened again.

(R9) Who Performs Changes and When: it has to provide facilities to identify an actor who performs changes and when the changes are made.

Based on these requirements, we compare EMFStore, EDAPT, and DiCoMEF in Table 1. Indeed, EDAPT provides a facility to create a composite change from a set of primitive changes, but it does not support creating a composite change from other composite change(s).

For the rest, we define an operation trace ω as the complete documentation of a transformation step $M'/MM = M/MM \gg \omega$ where M'/MM is the new model obtained after application of operation trace ω — *M* denotes a model or a meta-model, that doesn't matter anymore. And a history could then be defined as a sequence $M/MM \gg \omega_1 \gg \omega_2 \gg \omega_3 \gg \omega_{...}$. A trace provides both the information about the precondition and the postcondition of operations.

Figure 5 shows the history meta-model of DiCoMEF. We did not show a user model element in Figure 5 for the sake of simplicity. The *Create* operation creates a model element in the context of a container element. *Delete* operation deletes an existing

model element from its parent element. *Move* operation changes the container of an element. *Add* operation adds a model element (data values) to a list of elements. *Remove* operation removes a model element from the collection. *MoveIndex* operation changes the index of an element in a collection. *Set* operation updates a value of a single-valued attribute or reference. Each operation step has been formally defined as a transition between a state before and a state after (denoted by the $'$ superscript). Definitions of *Create* and *Delete* operations are provided below, the other operations could also be defined seamlessly using the formalization defined in section 3.

Create Operation: Create operation creates objects in the context of a container.

$$\underline{\mathrm{M/MM} \gg \mathrm{create}(e_1, r, e_2, i) \gg \mathrm{M'/MM}}$$

$$\begin{aligned}
& r \in E_R \wedge e_2 \in E'_{OB} \\
& \wedge [\![\mathsf{type}(r)]\!]'_c = [\![\mathsf{type}(r)]\!]_c \cup \{e_2\} \\
& \wedge \left(\tau(e_1) = \mathsf{owner}(r) \vee (\tau(e_1), \mathsf{owner}(r)) \in \rho_{eSuperTypes}{}^{\star}\right) \\
& \wedge [\![r]\!]'_r = [\![r]\!]_r \cup \{(e_1, e_2)\} \\
& \qquad \wedge pos(e_2, [\![r]\!]'_r) = i \wedge [\![r]\!]'_r - i = [\![r]\!]_r \\
& \qquad \wedge [\![E_R.containment]\!]_a(r) = \mathbf{true}^{\dagger}
\end{aligned}$$

† since $E_R \in E_C$ (see 3.4) and $type(r) = E_R$ and $E_R.containement \in E_A$ and $\mathsf{owner}(E_R.containment) = E_R$, expression $[\![E_R.containment]\!]_a(r)$ denotes if the reference r must be considered as a containment or not. EMF attaches importance to the organization of the information in a strict containment relationship and many operations provided by the Ecore API depend on this hierarchy. For sake of simplicity, we define $\kappa(r) \triangleq [\![E_R.containment]\!]_a(r)$.

Example: `create(` $net, Nt.place, start, 1$ `)`

Delete Operation : Delete operation deletes an existing model element along with its contents (child elements) from its parent element.

$$\underline{\mathrm{M/MM} \gg \mathrm{delete}(e_1, r, e_2) \gg \mathrm{M'/MM}}$$

$$\begin{aligned}
& r \in E_R \wedge e_2 \notin E'_{OB} \wedge \kappa(r) = \mathbf{true} \\
& \wedge [\![\mathsf{type}(r)]\!]'_c = [\![\mathsf{type}(r)]\!]_c \backslash \{e_2\} \\
& \wedge \left(\tau(e_1) = \mathsf{owner}(r) \vee (\tau(e_1), \mathsf{owner}(r)) \in \rho_{eSuperTypes}{}^{\star}\right) \\
& \wedge [\![r]\!]'_r = [\![r]\!]_r \backslash \{(e_1, e_2)\}
\end{aligned}$$

Example: `delete(` $net, Nt.place, start$ `)`

4.2 Merging

Merging is a process of fusion of the ω^L and ω^C histories in such a way that conflicts are avoided. Some order of execution of operations could be imposed to facilitate merging. For instance, if ω^C must be executed before ω^L, then this would force ω^L to be rolled-back, next ω^C would be applied and finally ω^L could be reapplied. But this process is time consuming (e.g., rollback a one day work for a propagated change that renames an Eobject). Another option could preserve ω^L and next apply ω^C while there is no conflict. When a conflict is detected, ω^L is rolled-back and the scenario is reversed. A user can keep or drop some changes from ω^L when it is re-applied. But this ordering is also a time consuming process . The optimal option is to consider M/MM$\gg \omega^L \gg \omega^C$. In this strategy, if a conflict occurs while applying ω^C, changes in ω^L that caused the conflict are rolled back.. This last strategy has been chosen in DiCoMEF and relies on an indepth analysis conflicts and of the causal relationship between the rolled back operations and other traces in the histories. This is discussed in the next section.

4.3 Conflict Detection

Conflict detection techniques can be classified as either state-based or operation-based (Altmanninger et al., 2009; Lippe and van Oosterom, 1992; Mens, 2002). State-based techniques do not capture the time sequence of operations that could be relevant to reconcile conflicts and are thus not acceptable for our objectives. Contrariwise, operation-based approach records changes whenever they occur and can capture refactoring changes (Mens, 2002).

A user might not be able to execute all operations propagated from a controller on his local copy. For instance, this could be the case when a user deletes a model element and a controller propagates a change which modifies the same model element (i.e., a delete operation and a set operation). Hence, the deleted element needs to be re-created (with the same UUI). DiCoMEF only rolls back the delete operation (recreate) and the dependent operations (a copy of these operations is stored in the delete operation). For instance, let an editor and a controller work on the same model instance described in section 4. The Editor deletes the *start* model element (instance of a *Transition* class) and sets the name of the *busy* model element (instance of a *Place* class) to *“active”*. A controller sends a change propagation to rename a *start* model element to *“begin”*. In order to apply the change propagation, a deleted element (*start*) must be re-created. During rolling back, DiCoMEF firstly creates the *start* model element and afterwards it reestablishes the relationships between the *start* and the *busy*, and the *idle* and the *start* model elements (see Figure 6). Rolling back only the delete operation is important, specially, if there are many changes performed by a user after deleting a model element.

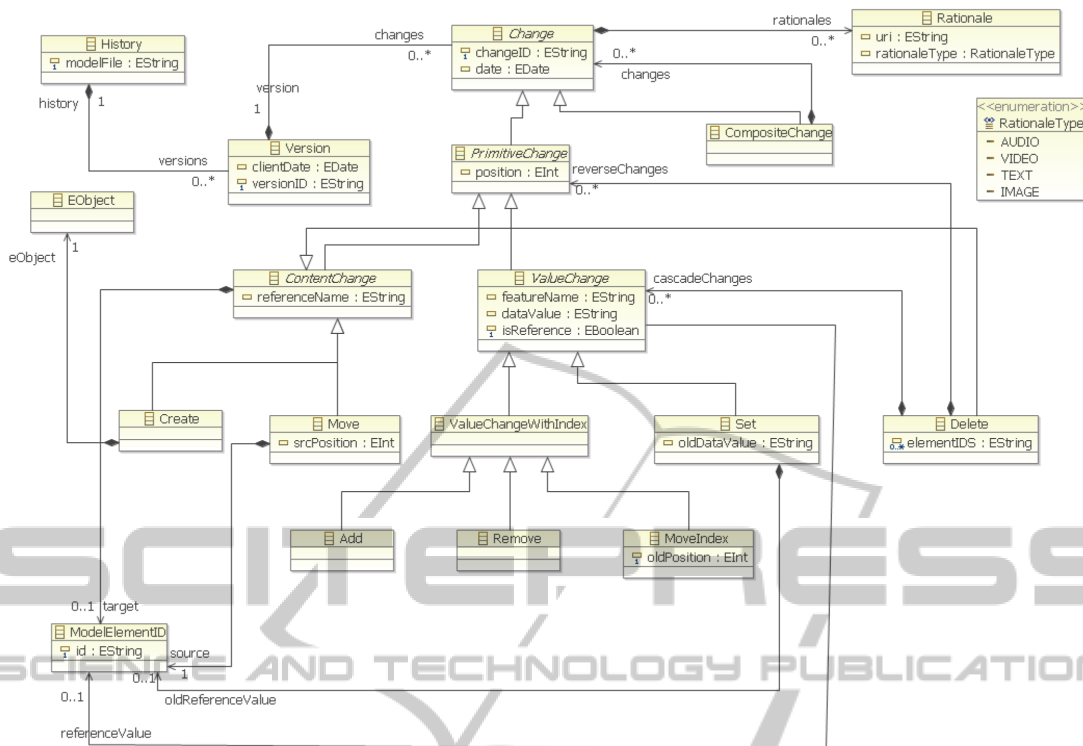

Figure 5: History Meta-model.

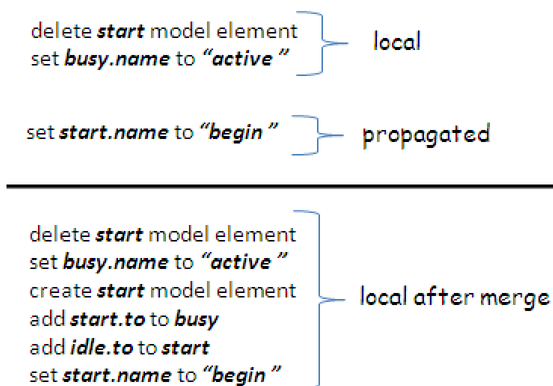

Figure 6: Change propagation and local operations.

Rolling back all changes to re-create a deleted model element could be time consuming. Finally, it renames *start* to *"begin"*.

We employ the *Conflict* (Table 2 and 3) and *Require* (Table 4) relations to detect conflicts between H^C and H^L (Koegel et al., 2009). An operation ω_i^C is conflicting with another operation ω_j^L if the order of serialization of these operations affects the final state of the (meta)model (e.g., two *set* operations that rename an EObject differently) (Koegel et al., 2009). Besides, the execution of one of the operation could invalidate a precondition of another one. The $\succ$ operator shows that one operation is succeeded (directly or indirectly) by another operation. For instance, $\text{create}(e_1, r_1, e_2, i) \succ \text{delete}(e_3, r_2, e_1)$ and $\text{delete}(e_3, r_2, e_1) \succ \text{create}(e_1, r_1, e_2, i)$ gives different result. In the first case, both operations execute and the model element e_1 along with its child are deleted from the model. But, in the second case, the create operation cannot execute because the delete operation deletes the target model element e_1 and makes the precondition of the create operation invalid. Therefore, $create(e_1, r_1, e_2, i)$ and $delete(e_3, r_2, e_1)$ are conflicting operations. A composite operation ω_c is in conflict with another operation ω_1 if at least one of its member operation is conflicting with ω_1. The semantics of conflicts for other operations could easily be expressed as well.

Conflict relation calculates a set of conflicting operations. The level of severity of conflicts could vary based on the type of conflicts meaning that some conflicts need user interactions whereas other conflicts could be solved automatically (Koegel et al., 2009). *Hard conflicts* require a user interaction. *Soft conflicts* can be resolved automatically by employing some conflict reconciliation strategies. Tables 2 and 3 show the conflicting relation — ***h*** (resp ***s***) denotes a hard (resp soft) conflict.

An operation, ω_j, requires another operation, ω_i,

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 2: Conflicting relation (ordered-multivalued).

	Create	Delete	Move	MoveIndex	Add	Remove	Set
Create	s	h	s	s	s	s	
Delete	h		h	h	h		h
Move	s	h	h	h	s	s	
MoveIndex	s	h	h	s	s	s	
Add	s	h	s	s	s	s	
Remove	s		s	s	s		
Set		h					s

Table 3: Conflicting relation (unordered-multivalued).

	Create	Delete	Move	Add	Remove	Set
Create		h				
Delete	h		h	h		h
Move		h	h			
Add		h			s	
Remove				s		
Set		h				s

if and only if ω_i must be executed before ω_j so that the precondition of ω_j is entailed by the post-condition of ω_i. The *require* binary relation is transitive, but it is not symmetric. For instance, a create operation requires another create operation that creates a target model element (container).

$$\text{create}(e_3, r_2, e_1, j) \succ \text{create}(e_1, r_1, e_2, i)$$

Hence, the require relation is extended with that pattern: $(\text{create}(e_1, r_1, e_2, i), \text{create}(e_3, r_2, e_1, j))$. The relation is resumed in Table 4. There is a relation between the *require* and *conflict* relationships: if operation ω_1 requires ω_2 and ω_2 conflicts with ω_3, then ω_1 also conflicts with ω_3.

Meta-model adaptation could also lead to a precondition violation, for instance, a reference feature of a meta-model element could be deleted in a new version of meta-model that results in violation of precondition for Create, Set, Add, ... operations. In this case, both the instance model and its respective history model needs to co-evolve with meta-model. But model co-evolution and history migration are not in the scope this paper.

When hard conflicts occur, the DiCoMEF framework shows the conflicting operations to the user with all the required information and the rationale about them. Once the user has solved the conflict, the merging process can continue.

5 CONCLUSION AND FUTURE WORK

To fully benefit from DSM tools, it is important to ensure cooperation among DSML tools. DiCoMEF is a distributed model editing framework where users (i.e., coordinators, editors, and observers) own their own local copies and can work asynchronously by exchanging operation traces. Specifying EMF models and the semantics of operations performed on models is a necessary process to assure an unambiguous communication between actors. This approach allows actors to better understand the rationale behind the evolution of models and to detect conflicts.

Table 4: Requires relation.

	Create	Delete	Move	MoveIndex	Add	Remove	Set
Create	√						
Delete	√						
Move	√						
MoveIndex	√						
Add	√						
Remove	√				√		
Set	√						

Modifications management is important to have a converging model after concurrent operations. In DiCoMEF, modifications are managed by a controller (human agent). More importantly, a controller role is flexible meaning that it could be easily assigned to another member. This dynamic roles assignment could lead people to implement more elaborated strategies on top of DiCoMEF, i.e., a user can delegate his/her role to another person. Although using a controller to manage collaborative modeling may limit the scalability, it could be possible to implement different method engineering techniques (e.g., delegation mechanisms, pooling) and strategies on top of DiCoMEF to address the problem. This article has presented a formalization of the conflicts that can occur in concurrent histories as well as important requirements history based strategies must fulfil.

Although the result of this work is fully operational, the reconciliation process could take place in the concrete syntax editors and meta-model semantics (i.e. OCL rules for instance) should be tackled in a future work. More advanced collaborative workflows should be also investigated and be defined on top of DiCoMEF.

DiCoMEF is implemented as an Eclipse plugin (54K LOC). The framework will be tested with master students during the academic year 2013–2014. Screenshots and other publications of DiCoMEF are found in https://sites.google.com/site/dicomef.

REFERENCES

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A Survey on Model Versioning Approaches. Technical

report, Johannes Kepler University Linz.

Blanc, X., Mougenot, A., Mounier, I., and Mens, T. (2009). Incremental detection of model inconsistencies based on model operations. In Eck, P., Gordijn, J., and Wieringa, R., editors, *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg.

de Lara, J. and Vangheluwe, H. (2002). Using atom as a meta-case tool. In *ICEIS'02*, pages 642–649.

Demeyer, S., Tichelaar, S., and Ducasse, S. (2001). FAMIX 2.1- the FAMOOS information exchange model.

Englebert, V. and Heymans, P. (2007). Towards more extensible metaCASE tools. In Krogstie, J., Opdhal, A., and Sindre, G., editors, *International Conference on Advanced Information Systems Engineering (CAiSE'07)*, number 4495 in LNCS, pages 454–468.

Gîrba, T., Favre, J.-M., and Ducasse, S. (2005). Using meta-model transformation to model software evolution. *Electron. Notes Theor. Comput. Sci.*, 137:57–64.

Gonzalez-Perez, C. and Henderson-Sellers, B. (2008). *Metamodelling for Software Engineering*. John Wiley, New York.

Graphical Modeling Framework (GMF) (visited: 2013). Graphical Modeling Framework. http://wiki.eclipse.org/Graphical_Modeling_Framework.

Herrmannsdoerfer, M. (2009). Operation-based versioning of metamodels with cope. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 49–54, Washington, DC, USA. IEEE Computer Society.

Kelly, S. (1998). Case tool support for co-operative work in information system design. In Rolland, C., Chen, Y., and Fang, M., editors, *Information Systems in the WWW Environment*, volume 115 of *IFIP Conference Proceedings*, pages 49–69. Chapman & Hall.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling. Enabling full code generation*. Wiley-IEEE Computer Society Pr.

Koegel, M. and Helming, J. (2010). EMFStore: a model repository for emf models. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *ICSE (2)*, pages 307–308. ACM.

Koegel, M., Helming, J., and Seyboth, S. (2009). Operation-based conflict detection and resolution. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 43–48, Washington, DC, USA. IEEE Computer Society.

Koshima, A., Englebert, V., and Thiran, P. (2011). Distributed collaborative model editing framework for domain specific modeling tools. In *ICGSE*, pages 113–118. IEEE.

Koshima, A. A., Englebert, V., and Thiran, P. (2013). A reconciliation framework to support cooperative work with dsm. In Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., and Bettin, J., editors, *Domain Engineering*, pages 239–259. Springer Berlin Heidelberg.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The generic modeling environment. In *Workshop on Intelligent Signal Processing*.

Lippe, E. and van Oosterom, N. (1992). Operation-based merging. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, SDE 5, pages 78–87, New York, NY, USA. ACM.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28:449–462.

Monperrus, M., Beugnard, A., and Champeau, J. (2009). A definition of “abstraction level” for metamodels. In *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, pages 315–320.

Monson-Haefel, R. and Chappell, D. (2000). *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Mougenot, A., Blanc, X., and Gervais, M.-P. (2009). D-praxis: A peer-to-peer collaborative model editing framework. In *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS '09, pages 16–29, Berlin, Heidelberg. Springer-Verlag.

Object Management Group (OMG) (2002). Meta Object Facility(MOF) Specification. http://www.omg.org/spec/MOF/1.4/PDF.

Pilato, C., Collins-Sussman, B., and Fitzpatrick, B. (2008). *Version Control with Subversion*. O'Reilly Media, Inc., 2 edition.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31.

Schmidt, K. and Bannon, L. (1992). Taking cscw seriously: Supporting articulation work. *Computer Supported Cooperative Work*, 1:7–40.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Taentzer, G., Ermel, C., Langer, P., and Wimmer, M. (2012). A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and Systems Modeling*, pages 1–34.

UML 2.0 superstructure (2011). *OMG Unified Modeling Language (OMG UML), Superstructure*. OMG. formal/2011-08-06.