Collaborative editing of EMF/Ecore meta-models and models conflict detection, reconciliation, and merging in DiCoMEF
Koshima, Amanuel; Englebert, Vincent

Published in:
MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development

Publication date:
2014

Document Version
Early version, also known as pre-print

Link to publication

Citation for published version (HARVARD):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 10. Dec. 2018
Collaborative Editing of EMF/Ecore Meta-models and Models: Conflict Detection, Reconciliation, and Merging in DiCoMEF

Amanuel Koshima, Vincent Englebert
PReCISE Research Center
University of Namur, Belgium
{amanuel.koshima,vincent.englebert}@unamur.be

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 reconciling 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.

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) 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 DICOME F

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). 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

Although these terms are also used by SCM programs, our framework does not rely on a central SCM.
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 w.r.t. 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 $V_1$.

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 MessageListenerImp that implements an IMessageListener 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 $R \subseteq A \times B$ 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 $(a, b_1) < (a, b_2) < (a, b_3)$ w.r.t $a$ 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 repre-
sented with pos function as follows \( pos(b_2, R(a)) = 1 \) and \([e_1, \ldots, e_2] - i \triangleq [e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n] \). If \( R \) is a binary relation \( \subseteq A \times B \), then we note \( R^- \) the inverse relation \( \subseteq B \times A \): \((a, b) \in R \iff (b, a) \in R^- \) and we note \( R'' \) its transitive closure, i.e. \( \{(a, b) \in R \lor \exists c : (a, c) \in R \land (c, b) \in R'\} \). \( 2^S \) denotes the powerset of a set \( S \) and \( A \rightarrow B \) 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 setting a containment boolean eOpposite link. Besides, a composition association between two references that are related to each other by bi-directional navigable association is modelled using an EAttribute, which has a name 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).

#### 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 \( \Sigma \) 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 \( E_{\langle A \times B \rangle} \). 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 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 \( \Sigma \). When the superclass is abstract, the inclusion is replaced with the equality operator. We bootstrap first the process by defining some sets:

\[
E_D = \{ EString, EInteger, \ldots \} \\
EString = \{ '/', ',', 'a', 'aa', 'ab', \ldots \} \\
EInteger = \{ EInteger.min, \ldots, -1, 0, 1, \ldots, EInteger.max \} \\
\]

\( E_D \) elements are data types. In EMF, a data type denotes 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), \( E_F \) (aka EPackage), \( E_R \) (aka EReference), \( E_I \) (aka EAttribute), \( E_E \) (aka EEnum), \( E_L \) (aka EEnumLiteral), \( E_O \) (aka EOperation), \( E_PA \) (aka EParameter), \( E_AN \) (aka EAnnotation), \( E_mE \) (aka ENamedElements), \( E_me \) (aka EModelElement), \( E_cl \) (aka EClassifier), \( E_fe \) (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.

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 EOb-

---

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

For sake of simplicity, we define owner(sf) \(= \rho_{cSF}(sf)\) and type(te) \(= \rho_{cType}(te)\) as respectively the owner of a structural feature and its type. \(\Sigma\) 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:
\[
E_C = \{Pl, Tr, Nt, Ne\} \\
E_A = \{Ne.name, Pl.tokens\} \\
E_R = \{Pl.to, Pl.from, Tr.from, Tr.to, \\
\]
an object diagram in Fig. 4) can be formalized as:

Example:

\[ \begin{align*}
\alpha_{\text{Class.name}} & = \{(\text{PL}, \text{`Place'}, \text{Tr}, \text{`Transition'}, \text{Nt}, \text{`Net'}, \text{Ne}, \text{'NamedElement'}, \text{Ne.name}, \text{'name'}, \text{Pl.tokens}, \text{'tokens'}, \text{Pl.to}, \text{`to'}, \text{Tr.from}, \text{from'}), \\
& \quad (\text{Nt}.\text{places}, \text{'places'}, (\text{Nt}.\text{transitions}, \text{`transitions'}))\}
\end{align*} \]

3.3 Instantiation

If \( MM \) is a meta-model, a model \( M \) compliant with \( MM \) (noted \( M/MM \)) is defined as a tuple \( (\Omega_C, \Delta_A, \Gamma_R, \Sigma_M) \) where each component is defined hereafter (\( E_{OB} \) denotes an infinite set of objects):

- \( \llbracket . \rrbracket_C : E_C \to 2^{E_{OB}} \): A class is modeled as a set of objects.
- \( \llbracket . \rrbracket_A : E_A \to (E_{OB} \times T^{-}) \) where \( T \) is the type of the attribute \( (T \in E_D) \). An attribute associates an object with values of type \( T \).
- \( \llbracket . \rrbracket_R : E_R \to (E_{OB} \times E_{OB}^{-}) \) Same for references, but with objects.

We define \( \tau(o) : E_{OB} \to E_C \) the function that maps an object \( o \) to its class \( c \in E_C \) such that \( o \in [\llbracket o \rrbracket] \land \neg \exists d \in E_C : \text{pESuperTypes}(d,c) \land o \in [\llbracket d \rrbracket] \).

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

\[ \begin{align*}
\text{Nt}.\text{places} & = \{\text{net}\} \\
\text{Pl}_C & = \{\text{idle, busy, dead}\} \\
\text{Tr}_C & = \{\text{start, kill}\} \\
\text{Pl}.\text{name} & = \{\text{idle', busy'}, \text{busy', busy'}, \text{dead', deadlock'}\} \\
\text{Tr}.\text{name} & = \{\text{start', start'}, \text{kill', kill'}\} \\
\text{Pl}.\text{tokens} & = \{\text{idle}, \text{idle'}, \text{busy, busy'}\} \\
\text{Pl}.\text{to} & = \{\text{start}, \text{kill}\} \\
\text{Pl}.\text{from} & = \{\text{dead, kill}, \text{busy, start}\} \\
\text{Tr}.\text{to} & = \{\text{start, busy}, \text{kill, dead}\} \\
\text{Tr}.\text{from} & = \{\text{start, idle}, \text{kill, idle}\} \\
\text{Nt}.\text{place} & = \{\text{idle, busy, dead}\} \\
\text{Nt}.\text{transition} & = \{\text{start, kill}\}
\end{align*} \]

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 \( \llbracket r \rrbracket \) 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. \( \text{MM}_{\text{Ecore}} \)). We could expect to observe the same property in our semantics framework of Ecore. Because of space constraints, we only present a partial definition of \( \text{MM}_{\text{Ecore}} \):

\[ \begin{align*}
\text{E}_C & = \{\text{E}_{\text{ne}}, \text{E}_{\text{P}}, \text{E}_{\text{C}}, \text{E}_{\text{R}}, \ldots\} \\
\text{E}_A & = \{\text{E}_{\text{ne}.\text{name}}, \text{E}_{\text{P}.\text{nsURI}}, \text{E}_{\text{P}.\text{nsPrefix}}, \\
& \quad \text{E}_{\text{C}.\text{instanceClassName}}, \text{E}_{\text{C}.\text{instanceTypeName}}\} \\
\text{E}_R & = \{\text{E}_{\text{P}.\text{eSubPackages}}, \text{E}_{\text{P}.\text{eClassifiers}}, \\
& \quad \text{E}_{\text{C}.\text{eStructuralFeatures}}\} \\
\text{E}_P & = \{\text{E}_{\text{core}}\} \\
\rho_{\text{SF}} & = \{(\text{E}_{\text{ne}}, \text{E}_{\text{ne}.\text{name}}), \\
& \quad (\text{E}_{\text{P}}, \text{E}_{\text{P}.\text{nsURI}}, \text{E}_{\text{P}.\text{nsPrefix}}, \\
& \quad \text{E}_{\text{P}.\text{eClassifiers}}, \text{E}_{\text{P}.\text{eSubPackages}}), \\
& \quad (\text{E}_{\text{C}}, \text{E}_{\text{C}.\text{instanceClassName}}, \\
& \quad \text{E}_{\text{C}.\text{instanceTypeName}}, \text{E}_{\text{C}.\text{eStructuralFeatures}})\}\]
used to define the semantics of a meta-model
flexive nature of Ecore. Moreover, the constructs are
itself meaning that our formalization support the re-
This process could be continued with the other con-
ne, instanceClassName, ‘instanceClassName’),
(EnE.instanceTypeName, ‘instanceClassName’),
(EnE.instanceTypeName, ‘instanceClassName’))
\[\alpha_{ne.name} = \{(E_{ne}, 'ENamedElement'),
(EnP, 'EPackage'), (EnC, 'EClassifier'),
(EnE.name, ‘name’), (Ee.nsURI, ‘nsURI’),
(Ee.nsPrefix, ‘nsPrefix’),
(EnE.instanceClassName, ‘instanceClassName’),
(EnE.instanceTypeName, ‘instanceTypeName’)\} \]
\[\rho_{E.type}(E_{ne}.name) = EString \]
\[\rho_{E.type}(E_{ne}.instanceClassName) = EString \]
\[\rho_{E.type}(E_{ne}.instanceTypeName) = EString \]
\[\rho_{E.type}(E_{ne}.nsURI) = EString \]
\[\rho_{E.type}(E_{ne}.nsPrefix) = EString \]
\[\rho_{E.type}(E_{ne}.eStructuralFeatures) = E_{ef} \]
\[\rho_{E.type}(E_{ne}.eClassifiers) = E_{c} \]
\[\rho_{E.type}(E_{ne}.eSubpackages) = E_{p} \]
\[\rho_{E.type}(E_{ne}.eClassifiers) = E_{c} \]
\[\rho_{E.type}(E_{ne}.superTypes(E_{c}) = E_{ne} \]
\[\rho_{E.type}(E_{ne}.superTypes(E_{p}) = E_{ne} \]
\[\rho_{E.type}(E_{ne}.classifiers(E_{p}) = E_{c} \]
This process could be continued with the other con-
structs 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 re-
flexive 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 con-
currently edited (meta)models need to be integrated
(merged), but most of the time they might not seam-
lessly work together as a result of inconsistent modifi-
cations (conflict). These conflicts should be identi-
cified 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 model-
ing 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 be-
fore their own operations. In DiCoMEF, the con-
troller 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, Di-
CoMEF supports a semi-automatic conflict reconcili-
ation strategy. Later, a user can send his/her lo-
cal modifications merged with the last release of the
model as a change request to the model controller. Di-
CoMEF 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 under-
stand changes during reconciliation process. It also
lets users to annotate rationale of changes with mul-
timedia files (i.e., audio, video, image, or text). Dur-
ing 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 cap-
ture the information denoted by the change operations
(create, delete and updates\(^{\text{3}}\)) of models: a model el-
ment can be created (or deleted), a value of a single
valued attribute or reference might be set. Besides,
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; Gürba
et al., 2005), a history meta-model based on a FAMIX
meta-model, is an artificial modification history. In-
deed, it transforms a snapshot model like UML or
FAMIX into a history rather than recording modi-
fications whenever they occur. It lacks preserving
their exact time sequences. EDAPT (Herrmannsdör-
erfer, 2009), previously called COPE, is a tool based
on EMF/Ecore meta-model that captures edit opera-
tions 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

\(^{\text{3}}\)Let’s note that read operations are not taken into con-

ideration.
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. 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 fulfill 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 to 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.

**(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 $M'/\text{MM} = M/\text{MM} \gg \omega$ where $M'/\text{MM}$ is the new model obtained after application of operation trace $\omega$. $M$ denotes a model or a meta-model, that doesn’t matter anymore. And a history could then be defined as a sequence $M/\text{MM} \gg \omega_1 \gg \omega_2 \gg \omega_3 \gg \omega_\ldots$. 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 $\text{Create}$ operation creates a model element in the context of a container element. $\text{Delete}$ operation deletes an existing model element from its parent element. $\text{Move}$ operation changes the container of an element. $\text{Add}$ operation adds a model element (data values) to a list of elements. $\text{Remove}$ operation removes a model element from the collection. $\text{MoveIndex}$ operation changes the index of an element in a collection. $\text{Set}$ 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 $'\superscript{r}$ superscript). Definitions of $\text{Create}$ and $\text{Delete}$ 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.

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

\[r \in E_R \land e_2 \in E_\text{OB} \land \{\text{type}(r)\}' = \{\text{type}(r)\} \cup \{e_2\} \land (\tau(e_1) = \text{owner}(r) \lor (\tau(e_1), \text{owner}(r)) \in P_e\text{SuperTypes}^*) \land [r]' = [r] \cup \{(e_1, e_2)\} \]

Table 1: Comparison of EMFStore, EDAPT, DiCoMEF.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMFStore</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>EDAPT</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>DiCoMEF</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

4Recently, EMFStore has had refactoring to reduce a coupling between parts of the implementation that captures history with rest of implementation.
\[ \land pos(e_2, [r]_r') = i \land [r]_r' - i = [r]_r \]
\[ \land [E_R\.containment]_a(r) = \text{true} \]
\[ \land \begin{align*} &E_R \in E_C \text{ (see 3.4)} \\ &\text{and} \ E_R\.containment \in E_A \text{ and} \ owner(E_R\.containment) = E_R \end{align*} \]
\[ [E_R\.containment]_a(r) \text{ denotes if the reference } r \text{ must be considered as a containment or not.} \]

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

\[ M/MM \gg \text{delete}(e_1, r, e_2) \gg M'/MM \]
\[ r \in E_R \land e_2 \notin E_{OB} \land \kappa(r) = \text{true} \]
\[ \land \begin{align*} &[\text{type}(r)]_c = [\text{type}(r)]_c \setminus \{e_2\} \\ &\land (\tau(e_1) = \text{owner}(r) \lor (\tau(e_1), \text{owner}(r)) \in p_{e\text{SuperTypes}^*}) \\ &\land [r]_r' = [r]_r \setminus \{(e_1, e_2)\} \end{align*} \]

**Example**: delete(net, Nt.place, start)

4.2 Merging

Merging is a process of fusion of the \(\omega^L\) and \(\omega^C\) histories in such a way that conflicts are avoided. Some order of execution of operations could be imposed to facilitate merging. For instance, if \(\omega^C\) must be executed before \(\omega^L\), then this would force \(\omega^L\) to be rolled-back, next \(\omega^C\) would be applied and finally \(\omega^C\) could be re-applied. But this process is time consuming (e.g., rollback one day work for a propagated change that renames an Eobject). Another option could preserve \(\omega^L\) and next apply \(\omega^C\) while there is no conflict. When a conflict is detected, \(\omega^C\) is rolled-back and the scenario is reversed. A user can keep or drop some changes from \(\omega^L\) 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 \(\omega^C\), changes in \(\omega^L\) that caused the conflict are rolled back. This last strategy has been chosen in DiCoMEF and relies on an in-depth analysis of 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...
Figure 6: Change propagation and local operations.

diagram of change propagation and local operations

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 (re-create) 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 re-establishes 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. 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 $H^I$ (Koegel et al., 2009). An operation $o_i^C$ is conflicting with another operation $o_j^C$ 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, $\text{create}(e_1, r_1, e_2, i) \succ \text{delete}(e_3, r_2, e_1)$ are conflicting operations. A composite operation $\omega_e$ is in conflict with another operation $\omega_i$ if at least one of its member operations is conflicting with $\omega_i$. 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, $o_i$, requires another operation, $o_h$, if and only if $o_h$ must be executed before $o_i$ so that the precondition of $o_i$ is entailed by the post-condition of $o_h$. 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 $o_i$ requires $o_h$ and $o_h$ conflicts with $o_j$, then $o_i$ also conflicts with $o_j$.

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.
Table 2: Conflicting relation (ordered-multivalued).

<table>
<thead>
<tr>
<th></th>
<th>Create</th>
<th>Delete</th>
<th>Move</th>
<th>MoveIndex</th>
<th>Add</th>
<th>Remove</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>s</td>
<td>h</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>s</td>
<td>h</td>
<td>h</td>
<td>s</td>
<td>h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MoveIndex</td>
<td>s</td>
<td>h</td>
<td>h</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>s</td>
<td>h</td>
<td>h</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Conflicting relation (unordered-multivalued).

<table>
<thead>
<tr>
<th></th>
<th>Create</th>
<th>Delete</th>
<th>Move</th>
<th>MoveIndex</th>
<th>Add</th>
<th>Remove</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>h</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

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.

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 fulfill.

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


