D-Praxis : A peer-to-peer collaborative model editing framework

Alix Mougenot[1⋆], Xavier Blanc[2], and Marie-Pierre Gervais[1]

[1] LIP6, Paris Universitas, France
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022,
Université des Sciences et Technologies de Lille, France

Abstract. Large-scale industrial systems involve nowadays hundreds of developers working on hundreds of models representing parts of the whole system specification. Unfortunately, few tool support is provided for managing this huge set of models. In such a context of collaborative work, the approach commonly adopted by the industry is to use a central repository and to make use of merge mechanisms and locks.
In this article we present a collaborative model editing framework, peer-to-peer oriented, that considers that every developer has his own partial replication of the system specification and that makes use of messages exchange for propagating changes made by developers. Our approach has the advantage not to be based on a single repository, which is more and more the case in large-scale industrial projects.

1 Introduction

Global specifications of large scale systems are more and more composed of multiple models that are distributed in different locations [5]. During the whole system life cycle, those multiple models are continuously and concurrently edited by developers who work asynchronously on their local copy and who commit from time to time the result of their work to the rest of their team [13]. In such a context, the main problem is to keep the local copies and the global specification consistent. To face this well known problem, conflicts that may arise while developers commit their changes have to be identified and resolved [1].
While the approach commonly adopted by the industry for managing conflicts is to use a central repository and to make use of merge mechanisms and locks [15], another approach, more peer-to-peer oriented, considers that every developer has his own partial replication of the global specification and makes use of messages exchange for propagating changes [14]. The former approach seems to be more robust to deal with model consistency, the later one has the advantage not to be based on a single repository which is more and more the case in large-scale industrial projects.
Peer-to-peer approaches for collaborative editing already exist, which are dedicated to text or tree based documents (XML) [6]. To our knowledge, none of

⋆ This work was partly funded by the french DGA.

them deals with models (typed graphs). Thus our aim is to propose such an approach for models.
Reviewing peer-to-peer approaches for collaborative data editing, we concluded that, whatever the nature of data (e.g, text or tree based), all of them are based on two concerns. The first concern is the conflict identification that requires a clear definition of the data structure and data changes. The second concern is a definition of a protocol required to propagate the changes as well as a mechanism for resolving conflicts that may arise after the propagation of changes.
In this paper, we deal with these two concerns by proposing D-Praxis, a peer-to-peer collaborative model editing framework. Our approach is based on message exchanges between developers for propagating changes. Conflict detection and resolution are done by each site when receiving changes messages.
This article is structured as follows. Section 2 deals with conflict identification and proposes formal specifications for models, model changes and collaborative model editing. Thanks to those specifications, a validation of conflict identification is presented. Section 3 presents the change propagation protocol and the mechanism that has been selected in order to resolve conflicts. Section 4 presents the implementation we have done, which is a decentralized peer-to-peer UML editor. Section 5 then presents related work and the last section presents our conclusion.

2 Conflicts identification

In this section, we present the formal definitions required for conflict identification. Since our concern is collaborative model editing, we propose four basic definitions: models, groups, sites and views (section 2.1). Based on them, we then define change semantics (section 2.2) and conflicts management (section 2.3). All our definitions are written in Alloy which is a lightweight specification language based on first-order relational logic [7].

2.1 Basic formal definitions: models, groups, sites and views

Various formalizations of models have been proposed in the literature [8, 2, 11]. For the sake of simplicity, we propose our own definition in Alloy that is compliant with those existing definitions but considers only a subset of MOF^{TM}[11]. In particular, we consider models to be a finite set of model elements where each model element is typed by a meta-class and can own values for properties and references (complex concepts such as opposite references, derived property and subset values are not supported). The definitions we provide are however sufficient for the purpose of this paper.

In this section, we use the following syntactic domains, represented as sets of atoms: `ME` for model elements, `MC` for meta-classes, `P` for properties, `R` for references and `V` for property values. Definition 1 presents the Alloy signature of the considered domains. In Alloy, a set of atoms is specified by a signature (**sig** keyword).

Definition 2 presents the Alloy specification of models where `mes` represents the set of model elements owned by the model (the keyword **set** means more than one), `class` the function that assigns one meta-class to each element, `valueP` the function that assigns one value to each property owned by model elements and `valueR` the function that assigns reference values. It should be noted that, in this formalization, all properties and references are single-valued.

A model can be included in another one if all its model elements, property values and reference values are included in the other one. Definition 2 gives, as an Alloy predicate, the formal specification of model inclusion. This definition will be used in the following sections.

Definition 1.

```
sig ME {} //model element
sig MC {} //metaclass
sig P {} //property
sig R {} //reference
sig V {} //value
```

Definition 2.

```
sig Model {
   mes: set ME, //elements of the model
   class: mes -> one MC, //metaclass
   valueP: (mes -> P) -> one V, //property values
   valueR: (mes -> R) -> one mes, //reference values
}

pred include[sub : Model , m : Model] {
   sub.mes in m.mes
   sub.class in m.class
   sub.valueP in m.valueP
   sub.valueR in m.valueR
}
```

Our definitions of group, site and view are based on [14, 12]. We consider that a group is composed of several sites that collaborate in the elaboration of one single global model. The global model is an intellectual artifact, it should not exist in the implementation. Each site has a partial view of the global model. Definition 3 presents the Alloy specification of a `Group` where `members` represents the set of sites and `globalModel` represents the global model that is shared between the sites. This definition also represents the specification of a `Site` where `view` represents the part of the global model that is viewed by the site.

Definition 3.

```
sig Group {
   members: set Site,
   globalModel : one Model,
```

```
}

sig Site {view: one Model,}
```

Definition 4 presents two Alloy facts that specify that (1) all the views of group members are models that are included in the global model of the group, and (2) all elements of the global model are in at least one view. According to those definitions, the intersection of two members' views can be empty or not. Figure 1 presents an example of a global model with two views. The left part of the figure presents the global model and the right part presents two views of it.

Definition 4.

```
fact allMembersOfAGroupeHaveAViewIncludedInTheGlobalModel{
   all g:Group, m:g.members | include[m.view,g.globalModel]
}

fact allModelElementOfAGlobalModelAreInViews {
   all g:Group | g.globalModel in g.members.view
}
```

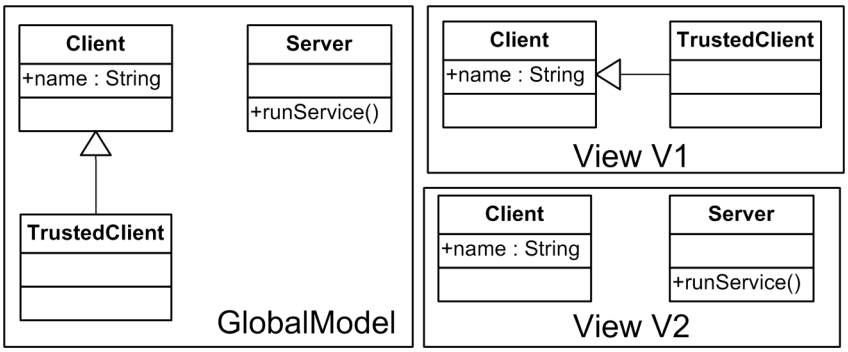

Fig. 1. A global model and 2 views

2.2 Change Semantics

In a group, each site can locally perform changes (in its view). Then, each site can decide to commit its local changes to the group. But changes made on a site view may impact the other site views. Thus it is necessary to define the semantics of a change and the impact it may have on other members of the same group.

In Alloy, such a semantics is specified thanks to predicates that have, by convention, two input parameters representing the same object but into two different

states; respectively before and after the occurrence of the changes.
Definition 5 presents the semantics of a model change and of a site change. A model `m` is modified, and becomes a model `m'`, if some of its elements, property or reference values are either modified or removed or if some new elements, properties or references are created (change of class are not allowed). A site `s` is modified, and becomes a site `s'` if its view is modified.

Definition 5.

```
pred modelChange[m , m' : Model] {
 m.mes != m'.mes || m.valueP != m'.valueP || m.valueR != m'.valueR
}

pred siteChange[s,  s':Site] {
   modelChange[s.view ,s'.view]
}
```

Definition 6 then presents the semantics of a group change. A group `g` is modified, and becomes a group `g'` if some of its sites are modified. Moreover, the specification defines that `g'`, the modified group, contains all the model elements, properties and references of `g` minus all those that have been removed, plus all the ones that have been newly created by the site changes.

Definition 6.

```
pred groupChange[ g,g':Group ]{
 some s:g.members | some s':g'.members {
  g'.members = g.members - s + s'
  siteChange[s,s']
  all me:ME |
   me !in s'.view.mes implies me !in g'.globalModel.mes
  all me:ME, p:P , v:V |
   ((me->p->v) in s.view.valueP && (me->p->v) !in s'.view.valueP)
     implies (me->p->v) !in g'.globalModel.valueP
  all me:ME, r:R , t:ME |
   ((me->r->t) in s.view.valueR && (me->r->t) !in s'.view.valueR)
     implies (me->r->t) !in g'.globalModel.valueR
} }
```

Definition 6 clearly specifies the semantics of group evolution. In particular, it defines that all elements deleted from a site view have to be deleted from the global model and then have to be deleted from all of the sites view that share them. For example, if the `Client` class is deleted from view `V2`, then it has to be deleted from the global model and therefore has to be deleted from the view `V1`.

2.3 Conflicts Management

The main known main difficulty for keeping consistency between a global model and its views is to deal with edition conflicts. In [10], Mens defines a conflict as

"A pair of operations that lead to an inconsistency is called a conflict". More formally, in our context, a conflict occurs when two sites of a group `g` perform changes that cannot be integrated into a single changed group `g'`.
We used Alloy in order to validate our conflicts identification. We used the following approach: (1) specify the changes with Alloy and (2) ask Alloy to find a configuration where there is a group `g` with two sites performing the changes and a changed group `g'` integrating both of them. If no configuration can be found by Alloy or if Alloy returns more than one configuration, then the two changes are said to be a conflict.
For sake of brevity, we present this approach in definition 7, but for only one kind of change because of space limitation. It consists of the addition of a new value (`v`) to the model element (`m`) for the property (`p`). This change applied on a site (`s`) returns the site (`s'`).

Definition 7.

```
pred addProp[s , s' : Site , m:ME , p : P , v:V] {
 s.view.mes = s'.view.mes
 m in s.view.mes
 s.view.class = s'.view.class
 m->p->v !in s.view.valueP
 m->p->v in s'.view.valueP
 no m':ME, p':P, v':V |
   m'->p'->v' in s.view.valueP && m'->p'->v' !in s'.view.valueP
 s.view.valueR = s'.view.valueR
}
```

Thanks to this definition, we have asked Alloy for an example of a changed group with two sites performing the change we just presented, on a same shared model element, on a same property but with two different values. This question, specified in the Alloy assert presented in definition 8.

Definition 8.

```
assert conflict1 {
 no s1,s1',s2,s2':Site , g,g':Group, m:ME, p:P  {
  s1+s2 in g.members
  s1'+s2' in g'.members
  some disjoint v1,v2:V |
   addProp[s1,s1',m,p,v1] && addProp[s2,s2',m,p,v2]
  groupChange[g,g']
 }
}
```

Asking this question to Alloy returns that no example can be provided. This means that Alloy cannot find a group `g'` that integrates those changes. Even if this answer cannot be considered as a proof, it is a validation that those changes are a potential conflict. Actually, as specified by definition 2, all model elements

can own a single value for each property. Therefore, the two different values added by the two changes cannot be integrated into the global model. It should be noted that, if we change definition 2 in order to support multi-value property, Alloy returns an example.

We have used the same approach to validate the following conflicts:

1. **different property values**: If two sites change the property value of one shared model element and if they provide different values.
2. **add and remove property value**: If one site changes a property value of one shared model element and if another one removes the value.
3. **different reference values**: If two sites change the reference value of one shared model element and if they provide different values.
4. **add and remove reference value**: If one site changes a reference value of one shared model element and if another one removes the value.
5. **delete a model element and add a reference value that targets it**: If one site deletes a shared model element and if another one adds a reference that targets it.
6. **delete a model element and add a reference value from it**: If one site deletes a shared model element and if another one adds a reference from it.
7. **delete a model element and add a property value to it**: If one site deletes a shared model element and if another one adds a property value to it.

It should be noted that this use of Alloy does not validate the fact that we have identified all possible conflicts. It only validates the conflicts that we have identified.

3 D-Praxis Protocol and conflict resolution

The second concern of any peer-to-peer collaborative editing is the propagation of changes and the conflict resolution. We then propose a protocol responsible for the propagation of changes and a mechanism to resolve conflicts between concurrent changes. Changing a model in our context means that a site modifies its own view. To represent such a change, a representation of models is required (subsection 3.1). Subsection 3.2 then explains how a global model is distributed among the different sites of a group. Subsection 3.3 explains how changes made by sites are propagated among other sites in order to update their views. Subsection 3.4 finally explains how conflicts are identified and resolved.

3.1 Representation of models

In [3], we propose to represent models as sequences of elementary operations needed to construct each model element. We reuse this approach and propose the six following elementary construction operations to represent any model. The six elementary operations we propose are inspired by the MOF reflective API [11]:

- **create***(me,mc)* creates a model element *me* instance of the meta-class *mc*. A model element can be created if and only if it does not already exist in the model (we use UUID to avoid two model elements with a same id);
- **delete***(me)* deletes a model element *me*. A model element can be deleted if and only if it exists in the model and it is not referenced by any other model element. When an element is removed, all its property values are automatically removed;
- **addProperty***(me,p,v)* assigns a value v to the property p of the model element *me*;
- **remProperty***(me,p)* remove the value, if any, to the property p of the model element *me*;
- **addReference***(me,r,met)* assigns a target model element *met* to the reference r of the model element *me*.
- **remReference***(me,r)* remove the target model element, if any, to the reference r of the model element *me*.

To illustrate this model representation, the left part of Table 1 presents a sequence of unitary actions corresponding to the class model of Figure 1 (Tx parameters are time stamp).

3.2 Representation of a global model and its views

As we presented in section 2, in D-Praxis, a group of sites collaborates to the elaboration of one single virtual global model. We consider the global model and the site's views to be represented by sequences of construction operations, respectively named the global sequence and the site sequences. All operations sequence are completely ordered thanks to a Lamport clock [9]. As each site's view is included in the global model, each site's sequence is a subpart of the global sequence. Moreover, as different site's views may share some model elements, their corresponding site sequences may share operations.
In D-Praxis, the global sequence is not to be completely owned by one particular site. However, its operations have to be distributed in the site's sequences. Table 1 presents the example model distributed within two sites. Each operation has a time stamp (the discriminator, which is the site's ID, is not represented here), represented by the last parameter of the operation. A cross in the table means that the site sequence owns a copy of the operation.

3.3 Changes propagation

As site views are represented by site sequences, changes made on site views are additions of new operations in the corresponding site sequences. Change propagation consists then in propagating those operations to the other sites, which may need to incorporate them in their own sequence to comply with the definition of the group semantics (see Definition 6).
In order to optimize change propagation and not to broadcast every change to all

GlobalModel	View V1	View V2
create(c1,class,T1)	X	X
addProperty(c1,name, 'Client',T2)	X	X
create(a1,attribute,T3)	X	X
addProperty(p1,name,'name',T4)	X	X
addProperty(p1,type, 'String',T5)	X	X
addReference(c1,attribute,a1,T6)	X	X
create(c2,class,T7)	X	
addProperty(c2,name,'TrustedClient',T8)	X	
addReference(c2,super,c1,T9)	X	
create(c3,class,T10)		X
addProperty(c3,name,'Server',T11)		X
create(o1,operation,T12)		X
addProperty(o1,name,'runService',T13)		X
setReference(c3,operation,o1,T14)		X

Table 1. Distributed representation of the example model (see figure 1)

the sites of a group, we propose to define subgroups of sites that share one same set of model elements (publish/subscribe strategy for each model element). When a change concerns a model element, it is only propagated to sites that share it. The following list defines our changes propagation protocol (conflict handling is discussed in the next section):

- **create***(me,mc)*: When a site adds a new model element in its view, the model element is automatically considered to be part of the global model because views are included in the global model (see definition 4). Moreover, this creation has no impact on other views (definition 6). Finally, as we use UUID, two sites cannot create two different model elements with a same id. This operation is not propagated.
- **delete***(me)*: When a site deletes a model element from its view, the model element is automatically considered to be deleted from the global model and from views that share it (definition 6). This operation has to be propagated to all the sites that share *me*.
- **addProperty***(me,p,v)*: When a site adds a new property value for a model element of its view, the global element is automatically changed as well as the views that share it (definition 6). This operation has to be propagated to all the sites that share *me*.
- **remProperty***(me,p)*: When a site removes the property value of a model element of its view, the global element is automatically changed as well as the views that share it (definition 6). This operation has to be propagated to all the sites that share *me*.
- **addReference***(me,r,met)*: When a site adds the reference value of a model element of its view, the global element is automatically changed as well as the views that share it (definition 6). This operation has to be propagated to all the sites that share *me* and *met*.

– **remReference***(me,r)*: When a site removes the reference value of a model element of its view, the global element is automatically changed as well as the views that share it (definition 6). This operation has to be propagated to all the sites that share *me*.

Change propagation does not prevent the conflicts identified in section 2.3. The next section presents how the propagated changes are integrated in the sequences of the sites that receive the changes and how conflicts are detected and resolved.

3.4 Conflict Resolution

Conflicts are detected and resolved by sites when they receive changes propagation. As said by Saito in [14], "Conflict resolution is usually highly application specific". Indeed, many strategies can be used to resolve them. The one we use is based on two principles: (1) the Lamport clock [9] and (2) the delete semantics of Praxis [3]. We consider that when two operations are in conflict, then the later one is kept and the former one is canceled unless it is a delete operation.
It is obvious to say that such a strategy may not be suitable to all developers' needs. Nevertheless, it has the advantage to be very simple, quite efficient and fits the objective of this work. Moreover, it should be noted that any other strategy can replace it.
The following list presents how conflicts (see section 2.3) are identified and resolved:

1. **different property values**: This conflict is detected if a site receives a new property value for one of its model element, which already owns another property value. Then, if the received operation is older than its own operation, the received operation is added to its site sequence, else it is ignored.
2. **add and remove property value**: This conflict is detected if a site receives a new property value for one of its model element where the property's value was already removed. This conflict is resolved like the previous one.
3. **different reference values**: This conflict is detected if a site receives a new reference value for one of its model element, which already owns another reference's assignment. This conflict is resolved like the previous one.
4. **add and remove reference value**: This conflict is detected if a site receives an addReference (resp remReference) for one of its model element, where this reference's assignment was already removed (resp added). This conflict is resolved like the previous one.
5. **delete a model element and add a reference value that targets it**: This conflict is detected if a site receives a delete operation of one of its model element that is locally referenced (definition 2 prevents model element to be deleted if they are referenced). Then, operations that remove all existing known references to the element from the current view are added to the site sequence (and won't be propagated) before the delete operation is added.
6. **delete a model element and add/remove a reference value from it**: This conflict is detected if a site receives a delete operation of one of

its model element, and this site has some operations that assign/remove references from this model element. This conflict is resolved by ignoring the reference changes that occur after the delete operation.

7. **delete a model element and add/remove a property value to it**: This conflict is detected if a site receives a delete operation of one of its model element, and this site has some operations that add/remove property values to this model element. Then, if some of those operations are older than the delete one, they are ignored.

4 Validation

4.1 Architecture

The validation we have done of our approach, named D-Praxis/UML, is a peer-to-peer collaborative editor for UML models. D-Praxis/UML is conceptually composed of the following three components that run on each developer site.
GroupMemberShip is the component that controls groups of developers. Thanks to this component, a developer can (1) create a new group, (2) join an existing group, (3) list the members of a group and (4) look at the view of a member in order to declare interest to some of its model elements. When a developer creates or joins a group, the GroupMemberShip component creates an empty UML model that corresponds to the developer's view. When a developer looks at the view of a member and declares interest at some of its elements, the GroupMemberShip component asks this member site for a propagation of operations corresponding the editing of those elements.
ChangePropagation is the component that controls messages propagation. Thanks to this component, a developer can change his view and commit his changes. The ChangePropagation component is responsible for setting the time stamps and propagating them to the interested developers. The ChangePropagation component is then responsible for the Lamport clock and for the interest groups that are used as routing tables for change propagation.
ConflictManager is the component that controls remote view editing. It receives changes and automatically integrates them in the developer's view. This component listens for change propagation and then is responsible for detecting conflicts and resolving them.

4.2 Sample scenario

As a validation scenario we present how two developers collaborate to the elaboration of a single global UML model. We consider that the group is composed of two developers: Bob and Alice. Thanks to the GroupMembership facility provided by D-Praxis/UML, Alice starts the creation of a new group. As a result, D-Praxis/UML creates an empty model on the Alice site. Alice then elaborates this model by adding the class `Client` of Figure 1. Those changes have time stamps that correspond to the Alice's clock. They don't have to be propagated

as there is no member who has declared an interest to them. Bob, who knows the IP address of Alice's site, uses the GroupMembership facility of D-Praxis/UML to join the group created by Alice. As a result, D-Praxis creates an empty model on the Bob's site. Bob then uses again the GroupMembership facility in order to look at Alice's view and to declare interest in the model elements representing the class Client and its attributes. As a result, the ChangePropagation component of Alice's site propagates corresponding operation to Bob's site. The first part of table 2 represents Alice's and Bob's views at this stage.
Alice and Bob then decide to change their view in parallel. Alice adds the `TrustedClient` class and Bob the `Server` class. Those changes have time stamps that correspond to Alice's and Bob's clocks. They don't have to be propagated as they do not target shared model elements.
Finally, Alice and Bob decide to change the `Client` class in parallel. Alice decides to make it abstract (addProperty(c1,abstract,'true')) while Bob decides to delete it (delete(c1)). As their changes target a shared model element, they are propagated. The Lamport clock states that the `delete` operation (B9) is before the `addProperty` one (A10). As we presented in section 2.3, those two changes make a conflict which is resolved by the rules we presented in section 3.4. Indeed, the delete operation is considered to be priority. As a consequence the operation remReference(c2,super,c1) is added to the Alice's view and the addProperty(c1,abstract,'true') is tagged as ignored in the Bob's view as shown in the last part of figure 1.

Alice	Bob
create(c1,class,A1) addProperty(c1,name, 'Client',A2) create(a1,attribute,A3) addProperty(p1,name,'name',A4) addProperty(p1,type, 'String',A5) addReference(c1,attribute,a1,A6)	create(c1,class,A1) addProperty(c1,name, 'Client',A2) create(a1,attribute,A3) addProperty(p1,name,'name',A4) addProperty(p1,type, 'String',A5) addReference(c1,attribute,a1,A6)
create(c2,class,A7) addProperty(c2,name,'TrustedClient',A8) addReference(c2,super,c1,A9) `addProperty(c1,abstract,'true',A10)`	create(c3,class,B7) addProperty(c3,name,'Server',B8) `delete(c1,B9)` create(o1,operation,B10) addProperty(o1,name,'runService',B11) setReference(c3,operation,o1,B12)
remReference(c2,super,c1,A11) delete(c1,B12)	`Ignore:`addProperty(c1,abstract,'true',A10)

Table 2. Bob and Alice views

4.3 Implementation

D-Praxis/UML has been developed on top of Eclipse as a set of Eclipse plugins[3]. The GroupMembership component is designed as a classic Group Membership framework but is very simple and does not support member disconnection. The ChangePropagation component is built on top of the EMF model editor that is used by developers for editing graphically their models. The ConflictManager component listens for propagation messages and, when it receives a message, automatically updates the site sequence and the corresponding model in the editor.

5 Related works

Collaborative editing has been already used for text and tree based documents but not for models [6]. Our approach follows principles defined by those approaches. In particular, it reuses the main idea of [4] which is to use operations for data and change representation.
In [12], the authors present a roadmap for building development environments that support multiviews editing and stress four points that have to be addressed: view initialization, view sharing, view change and view integration. Our proposal supports all of those four points by following a peer-to-peer decentralized approach.
Model editing conflicts have been presented in [10, 15]. Our approach identifies the same conflicts but provides a formal approach to validate conflicts identification.

6 Conclusion

In this article we proposed D-Praxis, a peer-to-peer collaborative model editing framework with which developers of a same team can collaborate to the elaboration of one single virtual global model. D-Praxis, is based on our previous proposal that is to represent model, and hence any view, by a sequence of construction operations. Changes propagation and conflicts detection and resolution are based on this representation.
As a first contribution, we have provided a formal validation of the identification of conflicts that may arise when changes are made on some views and lead to inconsistent model.
As a second contribution, we have provided a mechanism, peer-to-peer oriented, that handles changes propagation and conflicts detection and resolution. The change propagation protocol we propose is optimized in order to only target sites that share changed model elements. The conflicts resolution strategy we propose is based on an order computed by Lamport clocks and on the delete semantics of our model representation.

[3] http://meta.lip6.fr

Those two contributions have been implemented in D-Praxis/UML, peer-to-peer UML class editor. This implementation is currently stress tested by students of our university in order to better measure its advantages and its limits.
This kind of peer-to-peer oriented approach does offer solutions for dealing with collaborative work in a context where it is not possible to have a single repository.

References

1. E. W. Adams, M. Honda, and T. C. Miller. Object management in a case environment. In *Proc. Int'l Conf. Software Engineering (ICSE '89)*, pages 154–163, New York, NY, USA, 1989. ACM.
2. M. Alanen and I. Porres. A metamodeling language supporting subset and union properties. *Software and System Modeling*, 7(1):103–124, 2008.
3. X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Detecting model inconsistency through operation-based model construction. In Robby, editor, *Proc. Int'l Conf. Software engineering (ICSE'08)*, volume 1, pages 511–520. ACM, 2008.
4. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, 1989.
5. P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, et al. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University, ISBN 0-9786956-0-7, 2006.
6. C.-L. Ignat, G. Oster, P. Molli, M. Cart, J. Ferrié, A.-M. Kermarrec, P. Sutra, M. Shapiro, L. Benmouffok, J.-M. Busca, and R. Guerraoui. A comparison of optimistic approaches to collaborative editing of wiki pages. In *Proceedings of the 3rd International Conference on Collaborative Computing: Networking, Applications and Worksharing, White Plains, New York, USA, November 12-15, 2007*, pages 474–483, 2007.
7. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.
8. T. Kühne. Matters of (meta-) modeling. *Software and System Modeling*, 5:369–385, 2006.
9. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
10. T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.
11. OMG. Meta Object Facility (MOF) 2.0 Core Specification, Jan. 2006.
12. H. Ossher, W. Harrison, and P. Tarr. Software engineering tools and environments: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 261–277, New York, NY, USA, 2000. ACM.
13. D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large scale software development: An observational case study. In *Proc. Int' Conf. Software Engineering (ICSE '98)*, pages 251–260, 1998.
14. Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
15. P. Sriplakich, X. Blanc, and M.-P. Gervais. Supporting collaborative development in an open mda environment. In *ICSM*, pages 244–253. IEEE Computer Society, Sept. 2006.