A Semantically Rich Approach for Collaborative Model Edition

Jonathan Michaux, Xavier Blanc, Pierre Sutra, Marc Shapiro

A Semantically Rich Approach for Collaborative Model Edition

Jonathan Michaux,
Xavier Blanc, Pierre Sutra
LIP6
firstname.lastname@lip6.fr

Marc Shapiro
INRIA & LIP6
marc.shapiro@acm.org

ABSTRACT

We propose a novel approach and tool for collaborative software engineering and development. In model-based software engineering, the underlying data structure is a complex, directed and labeled graph. Collaborative engineering requires that developers be able to copy the graph, make independent changes, compare them, detect conflicts, and merge non-conflicting graphs. To support different collaboration and development styles requires a very flexible toolset. Worldwide, loosely-coupled development teams require the support of large-scale networks of users, possibly disconnected, in a decentralised fashion. No matter how the graph replicas evolve, they must eventually converge. We describe and evaluate C-Praxis, a tool that satisfies these requirements.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Design Tools and Techniques; I.6 [**Simulation and Modeling**]: Model Development, Applications

General Terms

Design, Reliability

Keywords

Collaboration, Consistency, Replication, Asynchrony

1. INTRODUCTION

Model Driven Engineering (MDE) projects involve increasingly larger and interrelated models [2, 4, 8]. Often, different projects share partial models. In this context, the current centralised and disjoint repositories constitute a severe limitation, as they do not provide any global consistency guarantees. Consider, for instance, the case where a model element is deleted from some repository even though it was referenced from another repository; this case is not currently supported, which results in an inconsistent model.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

Moreover, in a multi-project and multi-developer context, each one has its own policies and development patterns. A single, global collaboration policy cannot be assumed. Contrast the Linux project, with its hierarchical organization and well-defined access rights for individual developers, to other projects, which define access rights for individual model parts. Furthermore, some developers commit their changes quickly and often, in order to pre-empt possible conflicts with other developers, whereas others prefer to work in isolated, disconnected mode, and commit only when their task is ready. We argue that an efficient, reliable, flexible support tool is required for MDE projects, where developers collaboratively modify models throughout the entire development life cycles of different projects.

We present C-Praxis, a collaborative modeling framework that supports editing of models within a multi-project and a multi-developer context. C-Praxis is based upon Telex [1], a middleware layer designed for collaborative application software. Telex is based on a sound formal model that flexibly incorporates application semantics, supports many different collaboration styles, and provides well-defined consistency guarantees.

Telex is well suited to support collaborative MDE. Each *site* in the system stores a replica of the shared data. Each may independently perform update *actions* on its local replica. Sites synchronise by exchanging their actions in a peer-to-peer manner. The rate of synchronisation is a policy that can be set by each project. A developer may work in isolation or completely disconnected from the network. Telex is designed to support partial replication, whereby a site replicates only the data of interest to it; however, this article focuses on full replication, as partial replication is not currently functional. Telex supports the dynamic disconnection and reconnection of sites. Once reconnected, a site will be able to merge its own changes with those made by other sites. Conflicts may cause tentative changes to be aborted; however, Telex guarantees that all replicas eventually converge to a common, correct state.

The remainder of this paper is organized as follows: Section 2 details the challenges involved in designing a collaborative modelling environment. Section 3 presents our solution, C-Praxis. Section 4 explains how we used Telex in our approach. Section 5 presents our prototype implementation. Section 6 looks at some related work. We conclude in Section 7.

2. CHALLENGES OF COLLABORATIVE MODEL EDITING

Many modern software and system development projects are model-based [13]. To be able to manage their increasing complexity, multiple developers share models, and need the ability to work independently, in isolation. In such Collaborative Model Editing (CME) scenarios, developers may concurrently make overlapping or conflicting updates to a shared model. This section considers some of the issues in CME and outlines our approach.

CME provides each developer (or sub-group of developers) with a separate working copy, or *replica*, that he can modify in isolation. When replicas are concurrently modified, their states diverge. Such temporary divergence provides flexibility that is essential to large projects, as, for instance, it allows developers to explore different solutions, or to focus on a particular sub-system. However, in the long run, divergence should be resolved and replicas become mutually consistent, by reaching a consensus on the version of the model that will be adopted for the future.

Synchronising the replicas requires a common language to express model state and modifications thereof. The MOF standard that was designed to construct a model as a sequence of fine-grain operations [10] happens to be well-suited to express model changes and to exchange them between replicas.

Conflict resolution and consensus is a complex issue in general [12]. Perhaps the simplest solution is to retain one replica and discard the other. However, this defeats the purpose of CME. Another simple but inadequate approach is the Last-Writer-Wins (LWW) algorithm [6]. Here, each update is timestamped; when two updates are in conflict, the one with the highest timestamp (assumed to be newer) is retained, and the other one is discarded. This approach is very simple and is easy to understand, and it ensures that replicas eventually converge. However, it has severe drawbacks in the CME context, discussed next.

First, in LWW, the choice is arbitrary. In contrast, CME requires the ability to merge conflicts according to data semantics and under human supervision. Consider the following example. Developers Rita and James are editing a UML class HelloWorld, which is replicated onto Rita's and James' terminals. Rita changes the value of the class name property to GoodbyeMars. Concurrently, James changes the class name to AlohaJupiter. However, as the UML2 standard specifies that a class has a single, unique name [11], the two concurrent name changes conflict. Note that there is no single best solution to this conflict: only the developers can know whether the outcome should be HelloWorld, GoodbyeMars, AlohaJupiter, or some combination such as HelloWorld_GoodbyeMars_AlohaJupiter; furthermore, James, Rita and their supervisor may each have a different opinion. This illustrates that: (i) developers should have access to the multiple solutions of a given conflict, (ii) a developer should be able to pick a solution for his local view, and (iii) there should be a procedure for consulting the views and finally *committing* one of the solutions, which henceforth is considered authoritative.

The second major drawback of LWW is that the set of updates retained is not necessarily consistent. For instance, suppose that the class has a print method that outputs the class name as a string. Rita updates the string to be "GoodbyeMars" and James to "AlohaJupiter". With LWW, the timestamps may be such that the class name changes to GoodbyeMars while at the same time the string changes to AlohaJupiter! In contrast, CME requires that updates that belong together stay together. C-Praxis solves this issue in a very general way, by leveraging Telex's *constraints*, i.e., relations that assert that a given operation is *causally dependent, non-commuting, atomic* or *antagonistic* with respect to another given operation. This ensures that updates follow a correct workflow and that the models are well formed.

3. C-PRAXIS: AN APPROACH TO COLLABORATIVE MODEL EDITION

In this section we depict C-Praxis, our approach to CME. A representation formalism is introduced and then enriched with semantics which have been identified.

3.1 Representing a model as a sequence of operations

The MOF standard defines a model as a set of model elements, which are instances of meta-classes [10]. A model can be considered as a set of model elements that own values and refer to each other. Each model element is an instance of a meta-class that defines the properties it can own and the references it can have [10]. Blanc et al. show that *"Any model, regardless of its meta-model, can be represented as a sequence of operations performed to construct it, rather than by the set of model elements it contains"* [3]. These authors defined four elementary model building operations inspired by the MOF reflective API [10], later extended to six [9]:

create(*me*,*mc*) creates model element *me*, instance of meta-class *mc*.

delete(*me*) deletes model element *me*.

addProperty(*me*,*p*,*v*) assigns value v to the property p of model element *me*.

remProperty(*me*,*p* removes the value, if any, of property p from model element *me*.

addReference(*me*,*r*,*met*) assigns a target model element *met* to the reference r of model element *me*.

remReference(*me*,*r*) removes the value, if any, of reference r from model element *me*.

Each operation is meta-model independent. The latter four operations are *modification operations*.

The sequence below illustrates the use of these six model building operations to construct the class diagram depicted in Figure 1:

1. create(c1,Class)
2. addProperty(c1,name,'Apple')
3. create(c2,Class)
4. addProperty(c2,name,'Fruit')
5. create(a1,Attribute)
6. addProperty(a1,name,'variety')
7. addProperty(a1,type, 'String')
8. addReference(c1,attribute,a1)
9. create(a2,Attribute)
10. addProperty(a2,name,'nbPips')
11. addProperty(a2,type, 'int')
12. addReference(c1,attribute,a2)
13. addReference(c1,super,c2)

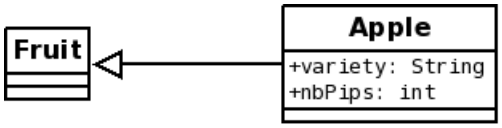

Figure 1: A simple UML class diagram

3.2 Communicating Change Information

The state of a model is represented by its corresponding sequence of operations. When a change is made to a model, a new operation is added to the sequence. These changes need to be added to the sequences of all other sites working on the same model as part of the effort to maintain consistency between all replicas. Indeed, two replicas are consistent if their corresponding sequences of operations are equivalent. In order for these changes to be taken into account by other sites, they are transmitted to Telex who propagates them. Because the system is totally replicated, every change regardless of its type is propagated to all sites.

3.3 Definition of Application Invariants

Application invariants are a set of rules that must be respected in order for models to remain correct. These invariants are properties that can be verified over the sequence of operations that represents the state of a model. Replicas that respect these invariants can be merged together.

The following set of meta-model independent invariants have been identified. They are based on the model representation described above and inspired by previous work [3, 9]:

- **Causality :** Given an operation sequence σ, if operation O is part of σ and affects model element *me*, then *create(me)* is part of σ and *create(me)* $<_\sigma$ O.
- **Delete Semantics :** Given a model element *me* and an operation sequence σ, if *delete(me)* is included in σ then there does not exist an operation O that targets *me* such that *delete(me)* $<_\sigma$ O.
- **Unique Identifiers :** Two model elements may not share the same unique identifier.

If any of these invariants are not respected, the resulting sequence of operations is incoherent and cannot be interpreted correctly. Without them, meaningless operations could be executed such as modification of a model element that has not yet been created or modification of an element after it has been deleted.

3.4 Conflict Identification and Resolution

Edition conflicts occur when a pair of developers concurrently modify the same parts of a model. These parts can be narrowed down to model elements. If a pair of operations affect the same model element(s), they are potentially in conflict. For example, two operations which modify the value of a property belonging to a certain model element are in conflict. To resolve this conflict, it must be decided which of the operations is to be taken into account, while the other will be ignored. Indeed, in this context there is no reason to attempt to merge the value from both property change operations - this does not make sense as the result will correspond to neither of the authors' intentions.

The following list presents the conflicts that have been identified:

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.

Note that the operations in conflicts 1, 2, 3 and 4 can coexist. It is their execution order that will affect the resulting state of the affected model elements. In order for these conflicts to be resolved, an order must be decided upon and reproduced on all sites.

However, when it comes to conflicts involving the deletion of a model element (as in conflicts 5, 6 and 7), resolution is more complicated. First of all, as stated in the delete semantics invariant earlier in Section 3.3, an operation on a model element cannot occur once the model element has been deleted. Therefore, if both operations are taken into account, the delete operation must always be performed last and therefore its effect will be taken into account (the element will be deleted). There is another way to resolve these conflicts which offers two valid solutions. In this approach, delete operation *delete(me)* is considered to be *antagonistic* with all modification operations on model element *me*. The first solution is to omit the delete operation in which case the modification operations are taken into account and the model element is not deleted. The second solution is to omit all modification operations on *me* and to simply take into account the delete operation. These two approaches shall be further examined in section 5.2.

4. PRAXIS OVER TELEX

This section explains how we use Telex to support C-Praxis. We first review basic Telex concepts, then detail how we leverage them to efficiently support CME.

4.1 Overview of Telex

Telex [1] is a middleware specifically designed to support collaborative applications and disconnected work settings. More precisely, Telex handles replication, consistency, storage and access control, as well as collecting, transmitting and persisting operations. Telex detects conflicts between

concurrent operations, and computes high-quality conflict-free schedules. It also offers forward execution, rollback, checkpointing and consensus. Telex uses semantic information provided by the application to execute consensus. This novelty gives Telex flexibility when merging concurrent modifications - as opposed to classical approaches such as last-writer-wins.

4.1.1 Telex fundamentals

Collaborative applications using Telex share *documents*. A document is an abstract data type that links different sites together based on the principle that they share data. A collaborative application interacts with Telex by submitting *actions* and *constraints* on one or more documents. An action is a reified application-level operation. A constraint is a concurrency control statement between two actions. Constraints materialize either workflow or application-specific invariants. Table 1 details the available constraints in Telex as well as their semantics.

Each Telex site maintains an *Action-Constraint Graph* (ACG) The ACG contains all actions and constraints known at one site. When a change is made to a *document*, Telex sends new actions and constraints to concerned replicas which update their local graphs.

To execute remote or local actions, an application is provided with Telex *schedules*. A schedule defines the state of the documents which are locally replicated. Every Telex schedule is *sound*, i.e., consistent with the constraints, possibly across multiple documents. This provides a synthetic, high-level view of conflict resolution.

4.1.2 Replication and data accessibility

The replication strategy used in Telex is *optimistic replication*, this means that sites access documents without a priori synchronization. Modifications are performed locally then propagated to other Telex sites in the background. Thanks to optimistic replication, Telex supports disconnected activity, i.e., an application can update its local replica of a document while disconnected from the network. Upon reconnecting, Telex notifies the application with new updates.

4.1.3 Consistency and consensus

Due to optimistic replication, replicas may diverge. In order to bound divergence, Telex runs a consensus protocol between sites. This protocol is based on *proposals* which are schedules approved by applications. Each site decides on a preferred proposal amongst those generated from the local ACGs. Telex picks the longest common sound prefix that exists based on each site's preference. When Telex cannot find common elements, it makes arbitrary decisions in order to finish the process. In the resulting decision, every action is either aborted or committed, and non-commuting committed actions are ordered. The result is then given a name and made to persist. The consensus protocol ensures eventual consistency, i.e., replicas eventually agree on a correct, common state for each replicated document.

4.2 Supporting C-Praxis with Telex

4.2.1 Conflict Identification and Resolution

An edition conflict is detected by Telex when a site receives a remote action on a locally replicated model. Telex compares the keys associated with the new changes to those associated to all the changes from the local edition history. If any of the keys match, then Telex has identified a potential conflict. The keys associated with an action have therefore been made to correspond to the unique identifiers of the model elements affected by the operation. Hence if two actions share the same key then they target the same model element.

When Telex detects a conflict it calls the application by transmitting it the pair of actions that provoked the detection. The application must then determine how to resolve the conflict between these two actions. To do this, the application has the option of creating a constraint between the two actions and returning it to Telex.

For example if Telex detects that two concurrent *addProperty* operations A and B affect the same model element, it will inform the application. The application then checks to see if the same property is being modified by both operations. If so, then the application has the option of deciding on an order between the two actions by using the *not-after* constraint. Using the formalism described in Table 1, this can be written $A \rightarrow B$ which indicates that A never occurs after B, therefore operation B's result is taken into account. Conflicts one through four described in Section 3.4 can be resolved this way.

Let us now consider the case of conflicts 5 through 7 from Section 3.4 which involve *delete* operations. Both approaches described in section 3.4 can be achieved using Telex. The symbols used are taken from table 1.

Approach 1: Given model element *me*, and a sequence of operations σ, if operation *delete(me)* is in σ, then any modification operation *mod(me)* in σ is such that *mod(me)* $\rightarrow$ *delete(me)*. This approach strictly favours the effect of the delete operation.

Approach 2: Consider model element *me*, and a sequence of operations σ. If delete operation *delete(me)* is present in σ, then given any modification operation *mod(me)*, *delete(me)* $\leftrightarrows$ *mod(me)*. Given the antagonist constraint used, this approach offers two possible solutions. The deletion can be ignored, in which case the modification operations are executed. Alternativeley, the deletion is taken into account, and the modification operations are not even executed in order to save time. Approach 2 has been retained in our solution because it offers a choice and is more economical.

4.2.2 Choosing a Schedule

As seen in the previous section, Telex will guarantee that a set of invariants are respected in order to generate *safe* (executable) schedules.

Telex can often generate more than one schedule from the action-constraint graph that contains the model's building operations and associated constraints. However, one solution must be chosen in order to continue working. Telex lets an application *select* a schedule, increasing its chances of being committed and reproduced on all replicating sites.

In order to benefit from the ability to choose, two categories of selection criteria have been defined : *organisational* criteria and *high-level* criteria.

Organisational criteria are used to select schedules based on the organisation and preferences of the development team. They target schedules that favour productivity and ergonomics. For example, the *edition history length* criterion gives priority to the developer with the longest edition history on

Name	Notation	Meaning for scheduling	Meaning for consensus
NotAfter	$a \rightarrow b$	a is never after b in any schedule	a is not after b in agreed prefix
Enables	$a \lhd b$	b in a schedule implies a in same schedule	b commits implies a commits
NonCommuting	$a \between b$	N/A	Conflict: Agree on either $a \rightarrow b$ or $b \rightarrow a$
Atomic	$a \overset{\lhd}{\rhd} b$	a and b both execute or neither does	a and b both commit or both abort
Causal	$a \overset{\lhd}{\rightarrow} b$	b executes after a, and only if a succeeds	b commits implies a commits
Antagonism	$a \overset{\leftarrow}{\rightarrow} b$	Conflict: a and b never both in same schedule	a commits implies b aborts, and vice-versa

Table 1: Constraint types. (a, b are arbitrary actions)

a conflicting part of a model. This is achieved by selecting a schedule that takes into account the most operations by the said developer. The *hierarchical priority* criterion favours schedules that include operations by developers that are higher up in the team hierarchy. These decisions can be made automatic in C-Praxis.

High-level criteria is used to evaluate schedules against high level rules that are domain specific or context sensitive. For example, Blanc et al. developed a consistency checking engine in Prolog [3]. It takes sequences of Praxis operations as an input, and checks them against a set of rules that can be specified by the user. These rules can include meta-model conformance, or compliance to structural rules such as those found in the UML2 [11] specification. Based on the output of this kind of checking engine, a schedule can be selected and a conforming model state is therefore promoted.

5. IMPLEMENTATION

5.1 Implementation

Our prototype was designed as an application layer above Telex that is generic because it is editor independent. It receives modification events produced by a model editor and outputs modification operations to be executed on a model. Hidden inside this layer are the semantics added to the sequences of operations before transmitting them to Telex, as well as the conflict identification and resolution mechanisms. All communication with Telex is also managed by this layer. An abstraction layer has been delimited that provides an API and can be plugged into any model editor. We have produced an implementation using a JGraphX [1] based graph editor, as well as an implementation that is integrated into the Eclipse EMF [2] environment.

Figure 2 shows the general architecture of *C-Praxis* and how a model editor is integrated into the framework. Our prototype is composed of four main components : the *PraxisTransmitter*, *Local Constraint Checker*, and the *Praxis Controller*. *Praxis Constraints* contains all the clauses relative to detecting a constraint.

A listener transmits a model modification event to the listener adaptor which converts this event into one of the six Praxis actions. The Praxis action is then sent to the Praxis Transmitter. Before transmitting this new action to Telex, the Praxis Transmitter uses the Local Constraint Checker component to check for local constraints. Given a pair of actions A,B, *Praxis Constraints* will determine whether or not a constraint exists between these two actions. The Local Constraint Checker uses Praxis Constraints to compare a new action with previous actions two by two. If a conflict is

[1]http://www.jgraph.com/
[2]http://www.eclipse.org/modeling/emf/

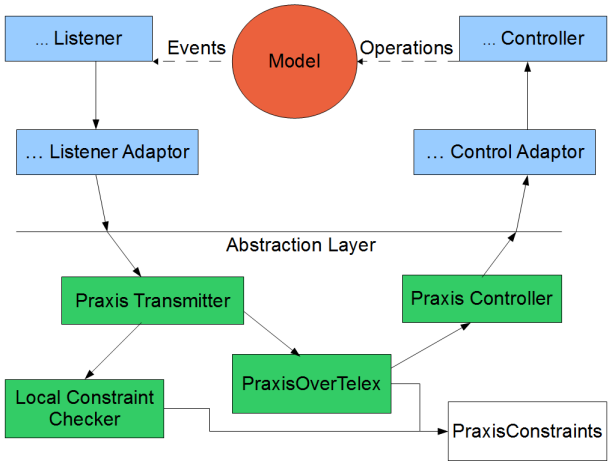

Figure 2: C-Praxis Architecture

detected, then the corresponding constraint is instantiated and submitted to Telex along with the new action.

On the other end, the Praxis Controller receives execution schedules from Telex. These are sent to the Control Adaptor which converts the schedule's Praxis actions into editor-specific operations that can be applied to the model.

5.2 Validation

In order to validate our approach, we present a scenario that showcases the advantages obtained thanks to a more flexible system of consensus and conflict resolution. This scenario is based on the example presented in Section 3.1, in which the sequence of operations required to create the fruit UML class diagram is given.

Consider two developers, Lucy and James. They have both begun working on a UML class diagram together. Lucy is the most productive of the two, and performs the following operations :

1. create(c1,Class)
2. addProperty(c1,name,'Apple')
3. create(c2,Class)
4. addProperty(c2,name,'Fruit')
5. create(a1,Attribute)
6. addReference(c1,super,c2)

James receives these operations, and the two developers have exact copies of the model. James now disconnects from the network. While James is disconnected, Lucy performs the following operations which add properties to the *Apple* class :

1. create(a1,Attribute)

2. addProperty(a1,name,'variety')
3. addProperty(a1,type, 'String')
4. addReference(c1,attribute,a1)
5. create(a2,Attribute)
6. addProperty(a2,name,'nbPips')
7. addProperty(a2,type, 'int')
8. addReference(c1,attribute,a2)

James on the other hand performs a *delete* on the *Apple* class. Both developers now have different views of the model which are not consistent with each other.

When James finally reconnects to the network, the software attempts to merge both replicas into a consistent state. Several edition conflicts are detected; indeed, as seen in Section 4.2.1 each modification operation on the *Apple* class performed by Lucy is in conflict with the deletion of the *Apple* class performed by James. An antagonism is the constraint expressed between these pairs of actions. The default schedule generated by Telex at the local level gives priority to operations performed by the developer who has the longest edition history. Hence, the default result will be that Lucy's modifications are taken into account, while James' delete is ignored. However, the developer is given the option of considering alternative schedules, including those that include James' delete operation and ignore Lucy's modifications. If the developer were to favour this option, then Lucy's modification operations are not executed by the model editor. This avoids useless computation time, while maintaining the modifications in the edition history for future reference or roll-backs.

6. RELATED WORK

D-Praxis [9], a peer-to-peer collaborative model editing framework, was the predecessor to C-Praxis. Indeed C-Praxis is a project created out of the desire to improve D-Praxis by adding disconnected operation and by using a sound distribution model. D-Praxis uses a last-writer-wins conflict resolution strategy based on a Lamport Clock [7]. Because it does not require the execution of lengthy consensus algorithms, a gain in performance is achieved, but at the cost of reduced flexibility for the end-user. D-Praxis supports partial replication of model data, which was not achieved with C-Praxis due to the design and immaturity of Telex.

CVS [2, 4, 8] is a centralised approach in which a central server stores and synchronises relevant data. Data is not associated to semantic information so merging conflicting modifications on data requires careful handling by the end user and can often be complicated.

7. CONCLUSION

C-Praxis is a collaborative modeling framework that allows asynchronous changes to be made on models while guaranteeing that a consensus is reached between replicas in the long run.

Through the use of the Telex middleware and its optimistic replication strategy, developpers can work while disconnected. For instance, a developer may wish to continue working without access to the network. Also, network connections are subject to faults and failures which would otherwise interrupt the development process.

Optimistic replication has advantages beyond network issues. Indeed, not only is it believed to be impossible to maintain absolute consistency between replicas at all times, it is not desirable [5]. With overly rigid consistency management, developers aren't given enough freedom to temporarily diverge from a consistent state, which is considered to be necessary in software development. *Flexibility* in model inconsistency management is a key feature provided by C-Praxis; it offers developers the advantage of choosing from a selection of valid model states before committing one. Furthermore, inconsistency management with Telex is a background process [1] which provides a more seamless experience.

In an effort to better evaluate the value of C-Praxis' flexibile consistency management, an empirical study would offer interesting insights into the usability of the tool and would provide a comparison with more convential software engineering techniques.

References

[1] L. Benmouffok, J.-M. Busca, J. M. Marquès, M. Shapiro, P. Sutra, and G. Tsoukalas. Telex: A Semantic Platform for Cooperative Application Development. In *Conférence Francophone en Systèmes d'Exploitation*, Toulouse, France, 2009.

[2] B. Berliner, P. Inc, and M. D. Blvd. Cvs ii: Parallelizing software development, 1990.

[3] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proc. Int'l Conf. Software engineering (ICSE'08)*, volume 1, pages 511–520, 2008.

[4] B. Collins-Sussman. The subversion project: buiding a better cvs. *Linux J.*, 2002(94):3, 2002.

[5] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[6] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[8] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proc. Int'l Conf. Automated software engineering (ASE '05)*, pages 204–213, New York, NY, USA, 2005. ACM.

[9] A. Mougenot, X. Blanc, and M.-P. Gervais. D-praxis : A peer-to-peer collaborative model editing framework. In *Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference, DAIS 2009, Lisbon, Portugal, June 9-11, 2009. Proceedings*, pages 16–29, 2009.

[10] OMG. Meta Object Facility (MOF) 2.0 Core Specification, Jan. 2006.

[11] OMG. Unified Modeling Language: Super Structure version 2.1, january 2006.

[12] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[13] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.