Annotation of Concurrent Changes in Collaborative Software Development

Claudia-Lavinia Ignat
INRIA Nancy-Grand Est
LORIA, Campus Scientifique, F-54506 Vandoeuvre-lès-Nancy, France
ignatcla@loria.fr

Abstract

Studies showed that in large projects the partition of software modules is limited and developers can contribute to any part of the code. In traditional software development tools such as CVS and Subversion users work in their local workspaces without being informed about concurrent modifications. This can lead to conflicting or redundant changes. We propose an awareness mechanism that informs users about the concurrent published changes by annotating the local project with these modifications. Users can continue working without integrating the concurrent changes being notified about the location of changes at different levels such as package, class, method and line. Users can also see the representation of the concurrent changes by consulting the annotations associated to code lines. We present the algorithms that implement our awareness approach.

1. Introduction

Most medium to large-scale projects involve multiple software developers that can be located in different places and might work on different time schedules. Traditionally, software projects are developed and maintained by means of a version control or a configuration management system such as CVS [2] or Subversion [3]. These tools allow developers to work in isolation on different or same parts of software and publish their changes at a later time.

Studies showed that in large projects the partition of software modules among developers is limited and developers can contribute to any part of the code [8]. While users work in isolation on their own copies of the source code and compile and test their changes before publishing them to the group, blind modifications might occur [9]. These blind modifications could lead to conflicts or redundant work. A conflict would, for instance, be generated if a developer modifies a method while another developer concurrently deletes that method. Two developers perform redundant work if they concurrently perform an identical task.

In order to avoid blind modifications developers should be informed as soon as possible about concurrent changes performed by other developers. Providing a user an understanding of who is working with him, what they are doing and how his own actions interact with theirs is called awareness [5]. We propose avoiding blind modifications by means of a suitable awareness mechanism.

Various tools were proposed to visualise awareness during software development by either augmenting existing views or constructing specialised views where human activities are combined with software artifact information [14, 4]. However, none of these approaches localises and represents changes performed by other users on software artifacts.

In this paper we present an awareness approach that avoids blind modifications by localising and representing concurrent changes. Integrating changes of other developers in real-time on a local copy of the source code is not feasible as this often leads to non-compiling code preventing the possibility to test code. We propose the annotation of source code with changes performed by other users. Annotations form an overlay model that is presented to users over their document view. Therefore, users can benefit from the awareness mechanism by means of annotations while continuing to change and test their code without actually integrating remote changes. In this paper we describe our awareness approach for changes that are published to the repository, i.e. users will be informed by means of annotations about concurrent changes committed to the repository and not yet integrated. The awareness mechanism about concurrent changes uncommitted to the repository relying on the same basic ideas as the awareness mechanism about concurrent committed changes is under current research and is not subject of discussion in this paper.

The main assumption of our novel awareness mechanism is that users are connected most of the time, even when they work in isolation. Since nowadays network connectivity is provided almost everywhere at the office, in mobile environments such as trains and planes, or out of the office in hotels or at home and it will continuously expand in the

near future, our assumption seems feasible. However, we can support disconnected work, but without providing any awareness mechanism.

The paper is structured as follows. In section 2 we present our envisaged annotation mechanism from the user and interface point of view. Section 3 presents the document model and the set of operations that we used in our annotation mechanism. In section 4 we describe the operation-based synchronisation mechanism that we used in our approach. Section 5 presents our solution for the envisaged annotation mechanism. Section 6 compares our proposition with related approaches. Finally, section 7 presents some concluding remarks and directions of future work.

2. Envisaged annotation mechanism

In this section we provide an example showing the envisaged annotation mechanism. For an easy understanding, we consider a very simple example involving two software engineers that collaborate on the source code of the same project stored on a central repository. The modifications they perform will overlap on the code of some common classes. Suppose that the first developer decides to remove the method isReal() from the class Integer illustrated in Figure 1 as he thinks that this method is not used throughout the project.

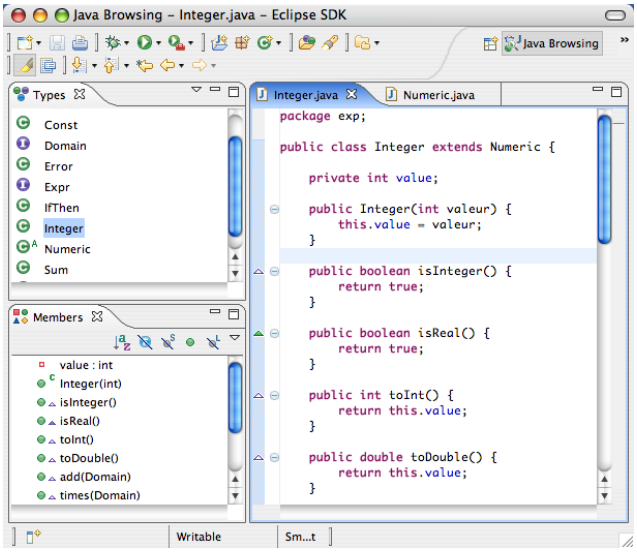

Figure 1. Initial document state

Concurrently, the second developer that uses the functionality of class Integer realises that the method isReal() that he uses should be corrected - it should return false as an integer should not be considered to be a real.

Let us analyse what awareness information is provided to the two developers depending on the order the two developers commit to the repository.

Suppose that the second user commits first to the repository the corrections he performed on the method isReal(). The first user who deletes the method isReal() will receive after the commit of the second user the awareness information presented in Figure 2. By means of a marker the user will be informed that the class Integer is concurrently modified as shown on the top left hand side window of the interface. In the right hand side window an annotation marker will indicate that a line was concurrently modified by another user. If the user examines details about the annotation, he will be informed that there is a conflict between his local change and the remote ones. For instance, the associated annotation in Figure 2 informs the user that the method isReal() locally deleted was modified by another user. In this manner, the user can decide to contact the other user or not to delete the method.

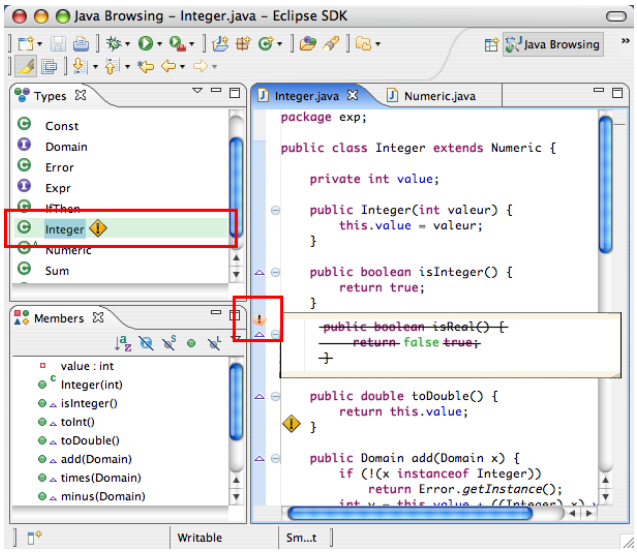

Figure 2. Interface at the first site after annotation of concurrent committed operations

Suppose now that the first user commits to the repository the deletion of the method isReal(). Let us analyse what happens at the site of the second user that modifies the method isReal(). After the first user commits his changes, awareness information about the changes performed will be sent to him and his local source code will be annotated as in Figure 3. In the right hand side window the user will be notified that the method isReal() is deleted by annotating the lines composing this method. The left hand side windows displaying the class hierarchy and the methods belonging to class Integer will highlight the fact that class Integer was concurrently modified and method isReal() was deleted.

In the next sections we describe our approach for developing the annotation mechanism presented in this section.

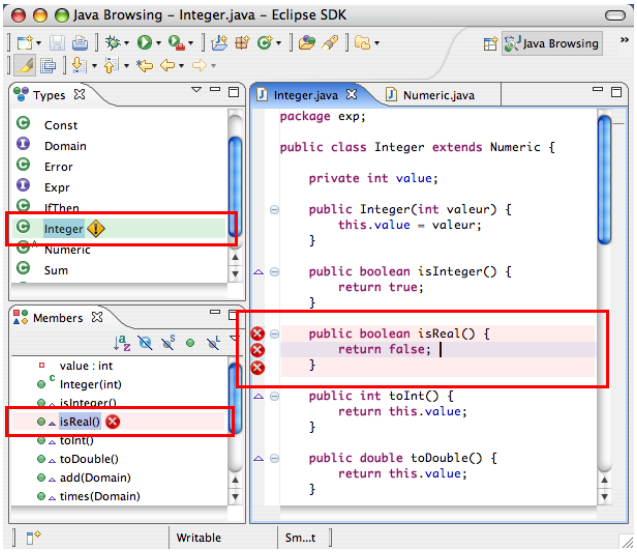

Figure 3. Interface at the second site after annotation of concurrent committed operations

3. Model of the document and set of operations

In order to capture changes at a low granularity level such as the character, we modeled the document as a sequence of characters and we represent changes performed on the document by means of the following two types of operations: *insert(p,c)* that inserts character *c* at position *p* and *delete(p)* that deletes the character at position *p*.

The linear structure of the document can be mapped to the structure of a source code document composed of packages, classes, methods and lines of code. Therefore, if a change was performed at a certain character of the document, we can annotate that a change was performed in the corresponding line of code, method, class or package.

4. Operation-based synchronisation over a shared repository

In this section we present the operation-based synchronisation mechanism over a shared repository by describing the basic methods that should be offered by a version control system. We then present the basic operational transformation mechanism in order to understand the basic steps of the update procedure.

4.1. Checkout, Commit, Update

The three basic methods supported by a version control system are: checkout, commit and update. A checkout method creates a local working copy of an object from the repository. A commit method creates in the repository a new version of the corresponding object by validating the modifications done on the local copy of the object. The condition of performing this method is that the repository does not contain a more recent version of the object to be committed than the base version of the local copy of the object, i.e. the last version from the repository that the user started working on. An update method performs the merging of the local copy of the object with the last version of that object stored in the repository.

In an operation-based version control system the repository stores a document version V_i by means of the set of operations representing the difference between V_{i-1} and V_i. The initial version V_0 is represented by the initial document state.

In the checkout phase, a request is sent to the repository to specify the version of the document that is intended to be checked out. The repository sends to the client the initial version of the document and the set of operations representing the difference between the requested document version and the initial document version. The client then executes the received list of operations on the received initial document state. It also sets the base document version to the requested version.

In the commit phase, a check is first performed as to whether the user can commit the changes to the repository. If the base version of the document in the local workspace is equal to the last version in the repository, a commit can be performed. Otherwise, an update is necessary before committing the changes. In the case that a commit is allowed, the repository should simply store the operations that were performed in the local workspace.

In the update phase, the repository sends to the local workspace a list of operations representing the delta between the latest version in the repository and the base version in the local workspace. Upon receiving the list of operations from the repository, the local workspace performs a merging algorithm to update the local version of the document. Consider the scenario illustrated in Figure 4 where the local user started working from version V_k on the repository but cannot commit the changes because meanwhile the version from the repository has been updated to version V_{k+n}. Let us denote by LL the list of operations executed by the user in their local workspace and by DL the list of operations representing the delta between versions V_{k+n} and V_k. Two basic steps have to be performed. The first step consists of applying the operations from DL on the local copy of the user in order to update the local document by integrating the changes included in V_{k+n}. The operations from the repository, however, cannot be executed in their original form as they have to be transformed in order to include the effect of all the local operations contained in LL before they can be executed in the user workspace. The

second step consists of transforming the operations in LL in order to include the effects of the operations in DL. The resulting list of transformed local operations represents the new delta to be saved in the repository.

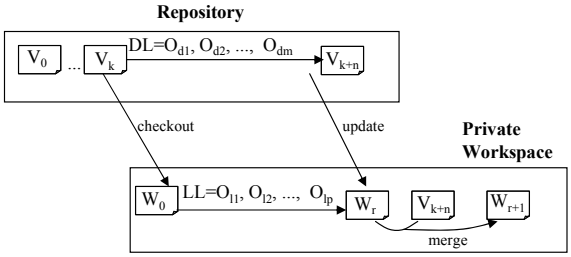

Figure 4. Updating Stage

4.2. Operational transformation

In order to illustrate the operational transformation approach we start by providing an example. Suppose the repository contains a class document with the content illustrated below. For simplicity, we represented only a small part of the document.

```
...
int concurencyContrl (...) {
...
```

Suppose that two users checkout this version of the document and perform some operations in their workspaces. $User_1$ performs the operation O_{11}=*insert(115,'o')*. We supposed that the method name *concurencyContrl(...)* corresponds to offset 100 in the source document. Therefore operation O_{11} is an operation of insertion of character *'o'* at the position 115 inside the document in order to correct the misspelling of the method name *concurencyContrl* to *concurencyControl*. Afterwards, $User_1$ commits the changes to the repository and the repository stores the list of operations performed by $User_1$ consisting of O_{11}. Further, assume that, concurrently, $User_2$ executes operation O_{21}=*insert(105,'r')* of insertion of character *'r'* at the position 105 inside the document in order to correct the misspelling of the method name *concurencyContrl* to *concurrencyContrl*. Before performing a commit, $User_2$ needs to update their local copy of the document. Operation O_{11} stored in the repository cannot be applied in its initial form on the local workspace of $User_2$, it needs to be transformed in order to include the effect of operation O_{21}. Because operation O_{21} inserts a character before the insertion position of O_{11}, O_{11} needs to increase its position of insertion by 1. This process of transformation of an operation to include the effects of another operation is called operational transformation. In this way, the transformed operation will become an insert operation of the character *'o'* at position 116, the result document being:

```
...
int concurrencyControl (...) {
...
```

Operational transformation mechanism was allowed to be performed in this case as the document states before the generation of O_{11} and O_{21} respectively were identical.

We explain next more formally the notion of operational transformation. Firstly, we present the notion of context [15] of an operation.

Definition 1 *The context of an operation O denoted as CT_O is defined as being the document state on which O is defined. Two operations O_a and O_b having the same context, $CT_{O_a} = CT_{O_b}$, are denoted $O_a =_{CT} O_b$.*

Definition 2 *An operation O_a is context preceding operation O_b denoted as $O_a \rightarrow_{CT} O_b$ if $CT_{O_b} = CT_{O_a} \cdot O_a$, i.e. the state of the document on which O_b is defined is equal to the state of the document after the execution of O_a.*

Next, we explain one of the basic mechanisms of the operational transformation approach [6], called inclusion transformation that we used in our algorithms. This type of transformation was illustrated in the previous example.

Definition 3 *The Inclusion Transformation - $IT(O_a, O_b)$ transforms operation O_a against operation O_b such that the effect of O_b is included in O_a. The condition of performing $IT(O_a, O_b)$ is that $O_a =_{CT} O_b$. If the result of $IT(O_a, O_b)$ is O'_a, then $O_b \rightarrow_{CT} O'_a$.*

4.3. Updating phase

In this subsection we present the operation-based merging algorithm used in the updating phase and that is also used by our annotation mechanism presented in section 5. The operation-based merging algorithm presented in this subsection is derived from the SOCT4 algorithm [16].

Consider the scenario illustrated in Figure 4 where we denoted by LL the list of operations executed by the user in their local workspace and by DL the list of operations in the repository that have to be applied locally to update the local document. Consider that DL and LL have the following structure $DL = [O_{d1}, ..., O_{d(i-1)}, O_{di}, O_{d(i+1)}, ..., O_{dm}]$, where $O_{d1} \rightarrow_{CT} ... \rightarrow_{CT} O_{dm}$ and $LL = [O_{l1}, ..., O_{l(i-1)}, O_{li}, O_{l(i+1)}, ..., O_{ln}]$, where $O_{l1} \rightarrow_{CT} ... \rightarrow_{CT} O_{ln}$. $O_{d1} =_{CT} O_{l1}$ as both lists LL and DL were defined on the same context namely the document state corresponding to version V_k. As mentioned in subsection 4.1 operations in DL need to be transformed

against operations in LL and operations in LL have to be transformed against operations in DL. Each operation in DL needs to be transformed against all operations in LL. When O_{d1} has to be applied on the local workspace, it is allowed to sequentially transform O_{d1} with respect to $O_{l1}, ..., O_{l(i-1)}, O_{li}, O_{l(i+1)}, ..., O_{ln}$ as

$O_{d1} =_{CT} O_{l1}$
$O^1_{d1} =_{CT} O_{l2}$ where $O^1_{d1} = IT(O_{d1}, O_{l1})$
$O^2_{d1} =_{CT} O_{l3}$ where $O^2_{d1} = IT(O^1_{d1}, O_{l2})$
...
$O^{n-1}_{d1} =_{CT} O_{ln}$ where $O^{n-1}_{d1} = IT(O^{n-2}_{d1}, O_{ln-1})$

When O_{d2} has to be applied on the local workspace, O_{d2} cannot be directly sequentially transformed against the list LL as O_{d2} is not contextually equivalent with the operations in LL as operation O_{d2} contains in its context operation O_{d1} while operations in LL do not. In order that transformations can be performed, operations in LL should include the effect of operation O_{d1}, the result being the list $LL^1 = [O^1_{l1}, ..., O^1_{l(i-1)}, O^1_{li}, O^1_{l(i+1)}, ..., O^1_{ln}]$ where

$O^1_{l1} = IT(O_{l1}, O_{d1})$
$O^1_{l2} = IT(O_{l2}, O^1_{d1})$ where $O^1_{d1} = IT(O_{d1}, O_{l1})$
...
$O^1_{ln} = IT(O_{ln}, O^{n-1}_{d1})$ where $O^{n-1}_{d1} = IT(O^{n-2}_{d1}, O_{ln-1})$.

Further, when O_{d3} has to be applied locally it needs to be transformed against operations in LL^1. In order that such transformations can be performed, operations in LL^1 need to be transformed against operation O_{d2} in the same manner as described above. The same process is in turn applied for operations $O_{d4}, \ldots, O_{dm}$. We can notice that at each step operations in DL and LL are transformed against each other. In order to support this process, the symmetric inclusion transformation between two operations O_a and O_b has been defined as follows:

Algorithm *symmetricInclusion(O_a,O_b):(O'_a,O'_b)* {
  O'_a*:=IT(O_a,O_b)*;
  O'_b*:=IT(O_b,O_a)*;
  return *(O'_a,O'_b)*;
}

In what follows we present the merge procedure. It takes as input arguments two logs, the remote log RL containing the operations from the repository and the local log LL containing the local operations. The merge procedure generates as output two other logs, the new remote log NRL and the new local log NLL, each of which is modified to include the effects of the operations in the other log. The new remote log NRL will contain the list of operations that should be executed sequentially on the current document state of the working copy in order to update it. The new local log NLL will store the list of operations which represent the delta between the new version and the old version in the repository and will have to be sent to the repository. The implementation of the merge procedure is given below.

Algorithm *merge(RL, LL):(NRL,NLL)* {
  for *(i:=1;i≤|RL|;i++)*
    for *(j:=1;j≤ |LL|;j++)*
      (RL[i],LL[j]):=symmetricInclusion(RL[i],LL[j]);
  NLL:=LL;
  NRL:=RL;
  return *(NRL,NLL)*;
}

5. Annotation mechanism

Additionally to the standard checkout, commit and update methods we offer an annotation mechanism of concurrent changes that is described in this section.

Each time a user commits changes to the repository, the repository informs the other users about the committed operations and user documents are annotated with the committed changes. Users can continue working without integrating committed changes, being informed by means of annotations about the concurrent changes performed. As showed in Figure 2, our annotation mechanism is meant to provide developers a quick overview about changes that occurred at a specific document part and who performed those changes. Changes performed by different users are depicted by different colours. For tracking user changes deleted parts of the document are marked as strike out and therefore they are not physically removed from the document.

Annotations should not modify the local document state and therefore local operations performed in the workspace do not have to take into account execution of annotation operations. We call an annotation operation a remote operation that will annotate the document. Annotations should be attached to the current document state. Therefore, for computing the proper location and effects of annotation operations in the current document, they have to take into account concurrent executions of local operations. The contents of the attached annotations to certain document parts should illustrate the integration of the annotation operations within the local document state corresponding to that document part. Therefore, merging between annotation and local operations has to be performed. In order to offer support for the visualisation of the deleted parts of the document, delete operations should mark for deletion the targeted characters and transformation functions used for merging should be adapted to do not take deletions into account. We used the tombstone transformation functions [13] for merging annotation operations with the local operations.

In order to deal with the annotation mechanism, each user workspace maintains the following data structures:

- LL the local log of operations containing the operations executed in the local workspace.

- ARL the annotation remote log containing the committed and non-updated operations from the repository that were used for the annotation process on the local document.

- ALL the list of local operations transformed against the ARL log. These transformations are performed in order to prepare further applications of annotations.

- $V.bv$ is the identifier of the base version of the local project, i.e. the latest version in the repository that was integrated in the local workspace.

We next describe the steps followed and how the above structures are updated in the case of the following main actions: checkout, execution of a list of local operations, commit, update of the local workspace and annotation of the local workspace. We mention that the actions of checkout, commit, update and annotation are atomic and therefore, no two of these types of actions independently called can interfere with each other. Moreover, if two actions of this type are sequentially sent by a certain site (client or server) to be applied on the remote site, they will arrive in the same order at the receiver site.

5.1. Checkout version $V.nv$

When a version of the project $V.nv$ is checked out, the following steps are performed:

- execute locally all received operations from the repository representing version *V.nv*
- *V.bv:=V.nv*
- *LL:=[]; ALL:=[]; ARL:=[]*
- *RL:=Repository.difference(V.nv,V.lv)*, where *V.lv* is the latest version in the repository
- call *annotate(RL)*

The local document is replaced with the version of the document from the repository and therefore, the list of operations received from the repository is locally executed on the initial document version received from the repository and the list LL is emptied. $V.bv$ is updated with the value of the checked-out version from the repository. The annotations are removed and therefore ALL and ARL are emptied. Afterwards, the repository computes and sends to the client the difference between the latest version in the repository $V.lv$ and the checked-out version $V.nv$. The client calls the annotation procedure described in subsection 5.5 with this difference passed as parameter.

5.2. Execution of a list of local operations

When a user executes locally the list of operations L_1, the following steps are performed:

- *append(L_1,LL)*
- *(NL_1,NARL):=mergeTTF(L_1,ARL)*
- *append(NL_1,ALL)*
- *ARL:=NARL*

List L_1 is appended to the list of local operations LL. In order to compute the location and contents of annotations, list L_1 is merged with list ARL. The transformed list of local operations NL_1 is appended to ALL. The annotation remote log ARL is replaced with $NARL$ obtained as result of merge to take into account the list L_1 of executed local operations. The base version of the local project $V.bv$ remains unchanged. The function *mergeTTF* has the same functionality as *merge* described in subsection 4.3 with the difference that the subprocedure *symmetricInclusion* uses the tombstone transformation functions.

5.3. Commit

A user is allowed to save his local version of the project to the repository only if his local version is up-to-date. If the local version is not up-to-date, the user has to update his local version of the project before committing his changes. When a new version of the project is committed to the repository, the following steps are performed:

- send to the repository the list *LL*
- *V.bv:=V.bv+1*
- *LL:=[]*; *ALL:=[]*; *ARL:=[]*

The list of local operations LL is sent to the repository and a new version of the project constructed from the list LL will be made available. The local base version is incremented. After the commit is performed, the list LL is emptied. The annotations are removed and therefore lists ALL and ARL are emptied, too.

5.4. Update

When an update is performed, the repository sends to the user the list RL representing the difference between the base local version of the user and the last available version of the project in the repository. The following steps are realised:

- *ALL:=[]*; *ARL:=[]*
- *(NRL,NLL):=merge(LL,RL)*
- *LL:=NLL*
- execute *NRL* sequentially on the local document state
- *V.bv:=V.lv*, where *V.lv* is the latest version in the repository

Annotations are removed and therefore lists ALL and ARL are emptied. The local list of operations LL is merged with the remote list of operations RL. The transformed list of local operations NLL will become the current local log. The operations in the transformed remote log NRL are sequentially applied on the local workspace. The base version of the local workspace is assigned the latest version in the repository. Note that for computing the document state as result of an update, the merge procedure described in subsection 4.3 is used. Delete operations physically delete characters in the document when they are executed and transformations against delete operations are performed.

5.5. Annotate the local document with a list of remote operations

Each time a user commits changes to the repository, the repository sends to the other connected users the list of committed changes. This list of remote operations denoted by RL is received by local users and used to annotate the document. These operations are not integrated on the local document, i.e. list LL remains unmodified as well as the base version of the document. The following steps are performed:

- *(NRL,NALL):=mergeTTF(RL,ALL)*
- use sequentially the operations in *NRL* to compute the annotations within the document
- *append(NRL,ARL)*
- *ALL:=NALL*

In order to compute the content of the annotations attached to parts of the document where concurrent changes occurred, the annotation operations in RL are merged with local operations in ALL. The operations in the transformed list NRL are used sequentially to compute an intermediary state of the local document that integrates annotation operations. The actual local document state does not change. The intermediary document state is scanned and the parts where annotation changes occurred are associated with local document lines. NRL is appended to ARL to keep track of all operations used in the annotation process. The list ALL is replaced by the transformed list $NALL$ to take into account the new list of operations RL used in the annotation mechanism.

6. Related work

CVS watches [2] permit users to subscribe for changes performed on an artifact and to be notified by email when a user announces by means of a command his intent to modify that artifact. However, watches require the use of email as an external tool for coordination in software development. In [7] the email notification mechanism is replaced with a lightweight notification mechanism called Elvin together with a tickertape tool where CVS messages are displayed and where developers can also chat with one another. Elvin and the tickertape are integrated in the CVS. These approaches do not provide a presentation mechanism of the changes performed.

VC^2 [10] is an awareness tool that can be integrated with existing version control systems. The file system is monitored for changes performed on documents. A user working on a document receives a notification when another user starts editing the same document. The user can then ask the other user for committing his changes. The second user might accept or reject the request.

In [4] a real-time awareness is provided for collaborative software engineering. Warning messages are used to notify developers about concurrent activity. Developers can afterwards consult the list of conflicts. Moreover, based on a selected conflict, a user can set watches for concurrently edited elements such that he is informed when the collaborator finished editing the element.

State Treemap [11] informs users about states of shared documents. Different states are defined to indicate when a copy is locally modified, when two copies of the document are modified and none of them is published yet or when a document copy is modified locally and some changes on that document were committed.

Palantìr [14] is an awareness tool for distributed software development based on the same principle as State Treemap, the main difference being that severity information that computes the amount of changes performed among documents is provided.

In the divergence metrics approach [12], metrics are not based as in Palantìr and State Treemap approaches on events triggered when the states of documents are changed, but they rather use information provided by operations that model concurrent changes. It is possible to compute the amount of concurrent changes performed on each document or an amount of conflicting/overlapping changes.

However, none of the previously mentioned approaches directly localises and presents the concurrent changes. Some of these approaches such as [2], [7] and [10] provide notifications about concurrent changes without any information on the changes performed. In [4] the notification mechanism allows users to consult conflicting changes that are listed in a separate document. In our approach concurrent changes are localised and annotated on the local document without any burden for users to set watches and analyse their results. The approaches in [11], [14] and [12] provide either a simple information that an artifact has been concurrently changed or a quantitative information about the concurrent changes performed on the same arti-

fact. However, no information about the location of changes is provided.

Some of the approaches described above present notifications or a qualitative measure of concurrent uncommitted changes. We presented an awareness approach only for concurrent committed changes. We are currently working on extending our approach for uncommitted changes.

7. Conclusions and future work

In this paper we presented an awareness approach that avoids conflicting or redundant concurrent modifications in collaborative software development. Our approach consists on annotating the local project with the concurrent changes. By means of annotations users are informed about the location and representation of concurrent modifications and can continue their work without integrating these changes in their local workspace. We expect that provided awareness information will generate group communication and auto-coordination between users in order to prevent conflicts and redundant work.

We presented the operational transformation algorithms for implementing the proposed annotation mechanism associated with a basic version control system. We developed a plugin for Eclipse [1] that implements the proposed annotation mechanism. We implemented a basic version control system where users can save and checkout versions of their shared projects and update their local projects with the published changes. Additional to a traditional version control system, our system informs users when new changes were published and these changes are used to annotate user local workspaces.

We are currently working on extending the awareness mechanism for concurrent uncommitted changes in addition to the committed changes. We plan to investigate the usability and the benefits of our approach by performing user studies.

References

[1] Eclipse - an open development platform. http://www.eclipse.org/.

[2] B. Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter Technical Conference*, pages 341–352, Washington, D. C., USA, Jan. 1990.

[3] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version control with Subversion*. O'Reilly & Associates, Inc., 2004.

[4] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2007*, pages 159–178, Limerick, Ireland, Sept. 2007.

[5] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW'92*, pages 107–114, Toronto, Ontario, Canada, 1992.

[6] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD'89*, pages 399–407, Portland, Oregon, USA, May 1989.

[7] G. Fitzpatrick, P. Marshall, and A. Phillips. CVS Integration With Notification And Chat: Lightweight Software Team Collaboration. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 49–58, Banff, Alberta, Canada, 2006.

[8] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW '04*, pages 72–81, Chicago, Illinois, USA, 2004.

[9] C.-L. Ignat, G. Oster, P. Molli, and H. Skaf-Molli. A Collaborative Writing Mode for Avoiding Blind Modifications. *Ninth International Workshop on Collaborative Editing Systems, GROUP'07, IEEE Distributed Systems online*, Nov. 2007.

[10] D. Machado, N. Preguia, C. Baquero, and J. L. Martins. VC^2 - Providing Awareness in Off-The-Shelf Version Control Systems. *Ninth International Workshop on Collaborative Editing Systems, GROUP 2007, IEEE Distributed Systems online*, Nov. 2007.

[11] P. Molli, H. Skaf-Molli, and C. Bouthier. State Treemap: an Awareness Widget for Multi-Synchronous Groupware. In *Proceedings of the International Workshop on Groupware - CRIWG 2001*, pages 106–114, Darmstadt, Germany, Sept. 2001.

[12] P. Molli, H. Skaf-Molli, and G. Oster. Divergence Awareness for Virtual Team Through the Web. In *Proceedings of World Conference on the Integrated Design and Process Technology - IDPT 2002*, Pasadena, California, USA, June 2002.

[13] G. Oster, P. Molli, P. Urso, and A. Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2006*, page 10, Atlanta, Georgia, USA, Nov. 2006.

[14] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantìr: Raising Awareness among Configuration Management Worspaces. In *Proceedings of the International Conference on Software Engineering - ICSE 2003*, pages 444–454, Portland, Oregon, USA, May 2003.

[15] H. Shen and C. Sun. Flexible Merging for Asynchronous Collaborative Systems. In *Proceeding of the Conference on Cooperative Information Systems - CoopIS 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 304–321, Irvine, California, USA, Nov. 2002.

[16] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies Convergence in a Distributed Real-Time Collaborative Environment. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2000*, pages 171–180, Philadelphia, Pennsylvania, USA, Dec. 2000.