Awareness of Concurrent Changes in Distributed Software Development

Claudia-Lavinia Ignat, Gérald Oster

Awareness of Concurrent Changes in Distributed Software Development

Claudia-Lavinia Ignat and Gérald Oster

INRIA Nancy-Grand Est, Nancy-Université, France
{ignatcla,oster}@loria.fr

Abstract. Traditional software development tools such as CVS and Subversion do not inform users about concurrent modifications while they work on shared projects in their local workspaces. We propose an awareness mechanism that informs users about location of concurrent changes at the level of class, method and line. Developers are provided with a preview of the concurrent changes made by other contributors on the project. They can continue to change and test their code without the need to integrate concurrent changes. We present the underlying algorithms of our proposed awareness mechanism.

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 system such as CVS or Subversion. 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 [1]. Therefore, 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. These blind modifications could lead to conflicts or to redundant work. Conflicts arise due to concurrent changes made to the same artifact or to dependent artifacts. For instance, a conflict is 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 the period of time between the conflict occurs and it is discovered, the conflict might grow and become difficult to resolve. Redundant work results into waste of time, energy and money.

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 [2]. 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 [3, 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. Our approach makes users aware about concurrent changes such that they can detect conflicts before committing their 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 the 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. 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, at home or in mobile environments such as trains and planes 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. We start by presenting in section 2 related awareness approaches about concurrent changes in software development. In section 3 we present our envisaged annotation mechanism. Section 4 presents the operation-based communication mechanism over a shared repository on which our approach relies. Section 5 describes our approach for realising the envisaged annotation mechanism. Finally, section 6 presents some concluding remarks.

2 Related work

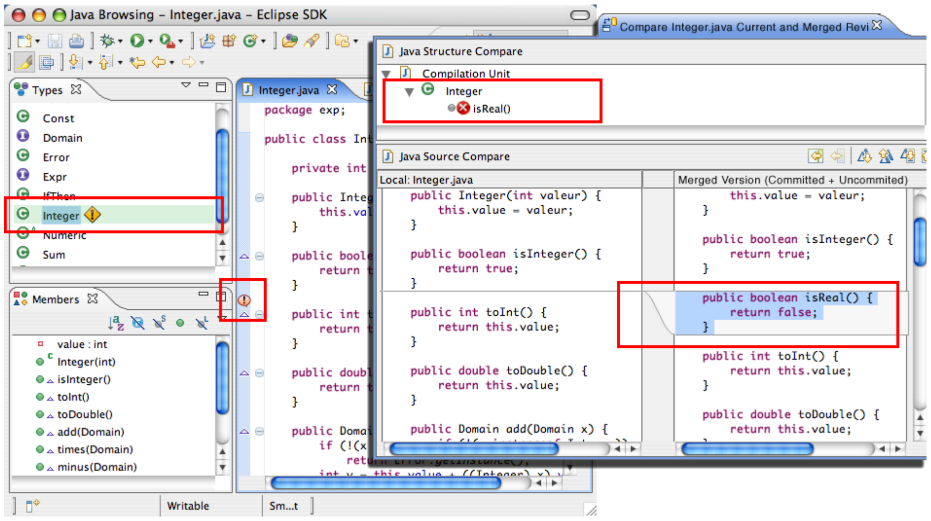
CVS watches permit users to subscribe for changes performed on an artifact and to be notified by email when a user announces by means of a command his intent to modify that artifact. However, watches require the use of email as an external tool for coordination in software development. In [4] a real-time 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 [5] informs users about states of shared documents that 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 [3] is an awareness tool for distributed software development based on the same principle as State Treemap. Additionally, it provides a severity information that computes the amount of changes performed on documents. In the divergence metrics approach [6], 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. In [4] the notification mechanism allows users to consult conflicting changes that are listed in a separate document. The approaches in [5], [3] and [6] provide either a simple information that an artifact has been concurrently changed or a quantitative information about the concurrent changes performed on the same artifact. However, no information about the localisation of changes is provided.

3 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 as he thinks that this method is not used throughout the project. 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.



```
package exp;

public class Int

    private int

    public Integ
        this.val
    }

    public boole
        return t
    }

    public int t
        return t
    }

    public doubl
        return t
    }

    public Domai
        if (!(x
            return Error.getInstance();
        int v = this.value + ((Integer) x).v
```

Local: Integer.java

```
    public Integer(int valeur) {
        this.value = valeur;
    }

    public boolean isInteger() {
        return true;
    }

    public int toInt() {
        return this.value;
    }

    public double toDouble() {
        return this.value;
    }

    public Domain add(Domain x) {
```

Merged Version (Committed + Uncommited)

```
        this.value = valeur;
    }

    public boolean isInteger() {
        return true;
    }

    public boolean isReal() {
        return false;
    }

    public int toInt() {
        return this.value;
    }

    public double toDouble() {
        return this.value;
```

Fig. 1. Interface at the first site after reception of concurrent operations

Let us analyse what awareness information should be provided to the first developer who deletes the method isReal(). He receives the changes of the second user and is provided with the awareness information presented in Figure 1. By means of a marker the user is 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 of the interface an annotation marker indicates that a line was concurrently modified by another user. The user can consult details about concurrent changes and is provided with a window where the difference between his changes and the concurrent changes performed on the document is presented. The user can see that the method locally deleted was concurrently modified by another user. In this manner, the user can decide to contact the other user or not to delete the method.

4 Operation-based collaboration over a shared repository

In this section we present the operation-based communication mechanisms over a shared repository for supporting the collaboration over code source documents.

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. By the base version of the local copy of the object we understand the last version of the object 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 represents a document version V_i by storing 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.

For the updating phase, an operational transformation (OT) mechanism such as the one presented in [7] is used. We present below some of the basic notions of OT that are used by our annotation mechanism.

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.*

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 $O_b \rightarrow_{CT} O'_a$.*

Definition 4. *The exclusion transformation - ET(O_a,O_b) transforms O_a against the operation O_b that precedes O_a such that the impact of O_b is excluded from O_a. The condition of performing ET(O_a,O_b) is that $O_b \rightarrow_{CT} O_a$. If the result of ET(O_a,O_b) is O'_a, then $O'_a =_{CT} O_b$.*

5 Annotation mechanism

Additionally to the standard checkout, commit and update methods we offer an annotation mechanism of concurrent changes. We first present the data structures a local workspace uses for dealing with annotations as well as the representation of operations. We next describe the algorithms for annotation and merging of concurrent changes.

5.1 Workspace data structures and operation representation

In a version control system users can simultaneously work on different versions of the shared project. Each time a user commits changes to the repository, the repository will inform the other users about the committed changes and user documents will be annotated with these changes. Uncommitted changes locally performed by users will be periodically sent directly to the other users and their documents accordingly annotated. Different types of markers will be used for annotation of committed and uncommitted changes.

We call an annotation operation a remote committed or uncommitted operation not yet integrated in the local document, but that is used to annotate the document. Since execution effect of an annotation operation does not change the document state, local operations performed in the workspace do not have to take into account its execution. However, for computing the proper localisation of annotation operations in the current document, they have to take into account concurrent executions of local operations.

In order to manage annotations, each user workspace maintains the data structure *(LL,RCL,RUL,bv)* where *LL* is the local log of operations executed in the local workspace, *RCL* is the remote log of committed and non-updated operations from the repository, $RUL[i]$ is the remote log of the non-committed operations performed by $User_i$ and *bv* is the identifier of the base version of the local workspace.

Operations in *RCL* and in $RUL[i]$ have to be transformed against the operations in *LL* in order to annotate the local version of the document. Transformations can be performed only if operations are defined on the same state.

In order to determine causality and concurrency between operations, operations have to be uniquely identified on the state of the document on which they were defined. We adopted the following representation for an operation: *O(type,position,content,bv,uid,sid,committed)* where *type* is the type of operation, *position* is the position in the document where the operation is applied, *content* is its content, *bv* is the base version of the local document, *uid* is the identifier of the user that generated the operation, *sid* is the sequence identifier of the operation with respect to *bv* and *committed* is a boolean indicating if the operation is committed or not.

A local operation has an associated base version at the moment of its generation. When the local version of the document is updated with new changes from the repository, local operations will be associated with the new base version of the document.

In order to capture changes at a low granularity level such as the character, we model 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)* to insert character c at position p and *delete(p)* to delete 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, a change made at a certain position of the document can be annotated at the corresponding line of code, method, class or package.

We define below the notion of causally readiness, precedence and concurrency that are used by our annotation algorithms.

Definition 5. *An operation O is causally ready for execution at a site if all committed operations with a smaller base version as well as operations with the same base version and same uid but with a smaller sid were received by that site.*

Definition 6. *A list of operations LL sequentially generated at a site where $LL[0] \rightarrow_{CT} ... \rightarrow_{CT} LL[m-1]$, m being the length of the list, is causally ready for execution at another site if $LL[0]$ is causally ready for execution at that site.*

Definition 7. *An operation O_1 is said to precede O_2 denoted as $O_1 \rightarrow O_2$ if one of the following conditions is satisfied: (a) $bv_1 < bv_2$ and $committed_1 = true$ (b) $bv_1 = bv_2$ and $uid_1 = uid_2$ and $sid_1 < sid_2$*

Definition 8. *Two operations O_1 and O_2 are said to be concurrent denoted as $O_1 \| O_2$ if $O_1 \not\rightarrow O_2$ and $O_2 \not\rightarrow O_1$.*

The structure *(LL,RCL,RUL,bv)* maintained at a local workspace is correspondingly updated when users commit or update their local workspaces. As soon as user $User_i$ commits a list RL of changes to the repository, these changes are sent by the repository to the other users. List RL is added to RCL maintained at their local workspaces and used to annotate their documents. Moreover, list $RUL[i]$ representing $User_i$'s uncommitted changes is emptied. When users perform local changes these changes are transmitted to the other users. A list of uncommitted changes RL made by $User_i$ is added to the list $RUL[i]$ maintained at the local workspaces of the other users and used to annotate their documents.

When a user $User_i$ performs an update, he notifies the other users about his update and sends them his new local log of operations. Upon receiving this message, the other users will update the list of uncommitted changes of $User_i$ with the new received list.

5.2 Annotation of concurrent changes

Principles

Suppose that the current state of the document is defined by the list of operations LL, where $LL[0] \rightarrow_{CT} \ldots \rightarrow_{CT} LL[m-1]$, m being the length of LL. Suppose that RL is the list of operations representing annotations that have to be applied on a document where $RL[0] \rightarrow_{CT} \ldots \rightarrow_{CT} RL[n-1]$, n being the length of RL. In order to apply the list of operations in RL as annotations on the local document, each operation in the list RL has to be transformed to be defined on the context of $LL[0]$. This can be done in two steps. The first step consists in transforming list RL into RL' where $RL'[0] \rightarrow_{CT} ... \rightarrow_{CT} RL'[n-1]$ and $RL'[0] =_{CT} LL[0]$. The second step consists in transforming $RL'[1]$, $RL'[2]$, $\ldots$, $RL'[n-1]$ into $RL''[1]$, $RL''[2]$, $\ldots$, $RL''[n-1]$ respectively such that $RL''[1] =_{CT} RL''[2] =_{CT} \ldots =_{CT} RL''[n-1] =_{CT} RL'[0] =_{CT} LL[0]$. Basically the second step transforms operations in RL into concurrent operations. Finally each annotation is applied on the document state and the corresponding lines of code are annotated.

Annotation of concurrent committed operations

Procedure *computeCommittedAnnotations* generates annotations from the list RL of committed operations. All operations in list RL are contextually preceding each other and they have all the same base version. As RL represents a list of committed operations not integrated on the local document version, the base version of this list is higher than the base version of the local document. If RL is causally ready for execution, it has to exclude the list RCL in order to be defined on the same context as $LL[0]$. Operations in the result list are then transformed to be each defined on the context of $LL[0]$ and then applied to annotate the document with the committed changes. The original list RL is then appended to RCL.

```
Proc computeCommittedAnnotations(RL)
    if (causallyReady(RL)) {
        ARL := ET(RL, RCL);
        ARL := transformIntoConc(ARL);
        applyAnnotations(ARL, true);
        append(RL, RCL);
    } else
        append(RL, TempRCL)
```

Procedure *transformIntoConc* transforms operations of list L to be defined on the context of definition of the operation $L[0]$. Each operation in L excludes the effects of the operations in L that precede it.

Procedure *applyAnnotations* annotates the positions of the document defined by the list of operations ARL. *flag* defines if operations are committed or uncommitted. Operations contained in ARL are transformed against the local list of operations LL.

```
Proc transformIntoConc(L):L'
    L' := L;
    for (i:=|L'|-1; i>1; i--)
        for (j:=i-1; j≥0; j--)
            L'[i] := ET(L'[i], L'[j]);
    return L';
```

```
Proc applyAnnotations(ARL, flag)
    for (i:=0; i<|ARL|; i++) {
        IT(ARL[i], LL);
        annotate(ARL[i], flag);
    }
```

Annotation of concurrent uncommitted operations

Procedure *computeUncommittedAnnotations* generates annotations from the list of uncommitted operations RL received directly from $User_i$. List RL contains contextually preceding remote operations having the same base version. If list RL is causally ready, it has to exclude all operations stored in RUL previously sent by $User_i$. ARL denotes the result of the transformation of RL. If $User_i$ worked on an older version of the document than the local base version, ARL has to be transformed to include the list of operations representing their difference. If $User_i$ worked on a more recent version of the document than the local base version, ARL has to be transformed to exclude their difference. In this way ARL and LL are defined on the same document state. Procedure *transformIntoConc* is then called to transform operations in ARL to be all defined on the generation context of the local log LL. Operations obtained as result of transformation are applied then to annotate positions of the local document where uncommitted changes occurred. List RL is then appended to the list of uncommitted operations of $User_i$. Due to space limitations, we do not include the algorithm that implements the procedure described above.

5.3 Merging of concurrent changes

Users can periodically consult the document state containing local changes, committed and/or uncommitted changes made by certain users. Using the list of local operations LL, the list of published operations not yet integrated in the local workspace RCL and the lists of uncommitted logs of other users RUL, merging of selected types of changes can be performed. We present the most general case that merges all concurrent changes.

```
Proc merge:H
    H := []; append(RCL, H);
    UL := []; append(LL, UL);
    for (i:=0; i<|RUL|; i++)
        append(RUL[i], UL);
    for (i:=0; i<|UL|; i++) {
        O := UL[i];
        j:=0; while (j<|H| and H[j]→O) j++;
        O' := transformSOCT2(O, sublist(H,j,|H|));
        append(O', H);
    }
    return H;
```

Procedure *merge* computes list H, the result of merging of LL, RCL and RUL. List RCL is first appended to H. Each operation O in LL and in RUL is integrated in turn into H. For the integration mechanism we use the SOCT2 [8] algorithm. Procedure *transformSOCT2* reorders the list of operations L such that operations that causally precede O are situated in L before operations that are concurrent with O. Afterwards O is transformed against operations that are concurrent with it and it is added at the end of L.

6 Conclusion

We presented an awareness approach that avoids conflicting or redundant concurrent changes in collaborative software development. By means of annotations users are informed about the location of concurrent modifications. Users can consult a document preview 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. We developed a plugin for Eclipse that implements the proposed annotation mechanism on top of our own operation-based version control system.

References

1. Gutwin, C., Penner, R., Schneider, K.: Group awareness in distributed software development. In: Proc. of the ACM Conference on Computer-Supported Cooperative Work - CSCW'04, Chicago, IL, USA (2004) 72–81
2. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proc. of the ACM Conference on Computer-Supported Cooperative Work - CSCW'92, Toronto, Ontario, Canada (1992) 107–114
3. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantìr: Raising awareness among configuration management worspaces. In: Proc. of the International Conference on Software Engineering - ICSE'03, Portland, OR, USA (2003) 444–454
4. Dewan, P., Hegde, R.: Semi-synchronous conflict detection and resolution in asynchronous software development. In: Proc. of the European Conference on Computer-Supported Cooperative Work - ECSCW'07, Limerick, Ireland (2007) 159–178
5. Molli, P., Skaf-Molli, H., Bouthier, C.: State treemap: an awareness widget for multi-synchronous groupware. In: Proc. of the International Workshop on Groupware - CRIWG'01, Darmstadt, Germany (2001) 106–114
6. Molli, P., Skaf-Molli, H., Oster, G.: Divergence awareness for virtual team through the web. In: Proc. of World Conference on the Integrated Design and Process Technology - IDPT'02, Pasadena, CA, USA (2002)
7. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. In: Proc. of the Conference on Cooperative Information Systems - CoopIS'02. Volume 2519 of Lecture Notes in Computer Science., Irvine, CA, USA (2002) 304–321
8. Suleiman, M., Cart, M., Ferrié, J.: Concurrent operations in a distributed and mobile collaborative environment. In: Proc. of the International Conference on Data Engineering - ICDE'98, Orlando, FL, USA (1998) 36–45