Gasper: A Collaborative Writing Mode for Avoiding Blind Modifications

Claudia Lavinia Ignat, Gérald Oster, Pascal Molli, Hala Skaf-Molli

Gasper: A Collaborative Writing Mode for Avoiding Blind Modifications

Claudia-Lavinia Ignat
LORIA-INRIA Lorraine
Campus Scientifique, BP 239
F-54506 Vandoeuvre-lès-Nancy, France
ignatcla@loria.fr

Gérald Oster, Pascal Molli and Hala Skaf
Nancy-Université, LORIA-INRIA Lorraine
Campus Scientifique, BP 239
F-54506 Vandoeuvre-lès-Nancy, France
{molli,oster,skaf}@loria.fr

ABSTRACT

In the last years network connectivity continuously expanded. However, existing collaborative environments were not designed to benefit from the fact that users are connected most of the time. For example, Wiki or version control systems allow users to work in isolation, but they tolerate blind modifications. For instance, users may concurrently perform the same task or they might work on obsolete versions of shared documents. We propose a novel writing mode for avoiding blind modifications by providing real-time information about group activities. Changes performed concurrently are filtered according to user privacy preferences and depicted in their local documents.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Asynchronous interaction, Synchronous interaction, Computer-supported cooperative work*

General Terms

Design, Human Factors

Keywords

CSCW, Collaborative writing, Blind modifications, Awareness, Synchronous interaction, Asynchronous interaction

1. INTRODUCTION

Collaborative writing is becoming increasingly common, often compulsory in academic and corporate work. The majority of all written work is produced collaboratively [7]. Writing journal papers and technical manuals, developing software code, and planning presentations are few examples of common collaborative writing activities.

Many definitions of collaborative writing exist [13]. In this paper, we consider the most commonly used one, in which, *collaborative writing is the process of two or more people working together to create a complex document.*

The major benefits of collaborative writing include reducing task completion time, reducing errors, getting different viewpoints and skills, and obtaining an accurate document [22, 16]. On the other side, collaborative writing raises many challenges ranging from the technical challenges of maintaining consistency and awareness to the social challenges of supporting group activities and conventions across many different communities.

The nature of the collaboration varies extensively [22] according to the degree of physical proximity of group members and synchronicity of writing activities. Group members working in various organisations can be located in different places and might work on different time schedules. Sometimes members work closely together and give immediate feedback to the changes of other users, and other times they work separately without quickly reacting to group contributions. Therefore, collaborative writing systems offer support to various working modes such as real-time and asynchronous interactions.

In real-time collaborative systems such as CoWord [20] and SubEthaEdit [1], changes performed by one user are immediately seen by other group members. These systems support reactive writing [13], where members react and adjust each others modifications without an explicit pre-planned coordination strategy. For instance, these systems could be used to offer features of face-to-face collaboration [8] when group members are located at different places. Brainstorming is an example of activity where users have to quickly propose and give feedback to other propositions in order to achieve a consensus. On the other side, collaboration might not be effective due to the fact that all changes are immediately visible by other members. Draft changes of a user might disturb the activities of other members. Users might not want to make public intermediate changes until they reach a final refined version of their contributions. For example, a group member writing in a foreign language would like to be able to carefully correct misspelling in his changes before annoying other members with unreadable contributions. Another example where real-time editing is not suitable is collaborative writing of software code. Developers have to work in their workspaces on their own copies of the source code without integrating in real-time partial changes of other developers.

GROUP '07, November 4–7, 2007, Sanibel Island, Florida, USA

Indeed, real-time integration of changes in the source code often leads to non-compiling code preventing the possibility to test their code.

According to Ellis et al. [8] groupware systems that do not support simultaneous activities are called non-real-time or asynchronous systems. In these systems, changes performed by users are not immediately transmitted and visible to other users. Users can interact over an extended period of time allowing users to connect and disconnect from the collaborative environment. One of the working mode that could be supported by asynchronous systems is isolated work. In this working mode, users can work in their workspace and decide when to initiate communication with the group, i.e. send and integrate changes. Therefore, asynchronous systems offer users the possibility of working even when they have no access to any network connexion, by allowing them to perform changes offline and publish these changes at a later time. When users work in isolation, they are not bothered with changes performed by other group members since they can decide when to integrate these changes. Furthermore, users can choose to send only completed work and in this way avoid to disturb other group members with partial changes. Version control systems such as CVS [5] and Subversion [2] are examples of asynchronous collaborative environments that are commonly used in software engineering. Developers work in isolation on their own copies of the source code. They can compile, test their changes and decide when to publish them to the group. In the same manner, developers can decide when to update their copies with changes published by other developers. However, working in isolation may generate blind modifications. Users perform blind modifications when they modify a document without being aware of concurrent changes. Blind modifications can lead to useless or redundant work. For example, useless work can occur if a user updates a section of a document while another user concurrently deletes this section. Two users perform redundant work if they concurrently perform an identical task.

The main issue addressed in this paper is how to prevent blind modifications occurring in isolated mode.

Blind modifications are caused by a lack of group communication or activity coordination. A well defined work process where tasks are clearly identified and assigned to different users prevents redundant work from occurring. However, collaborative writing processes are known to be non-linear and dynamic [4]. New tasks are often created and assigned during the collaboration. The redefinition of the process might occur when users are working in isolation. Therefore, two users can create a new task and start to perform this identical task in parallel. Improving communication among group members and their activities might resolve this problem. Unfortunately, due to the fact that users are distributed in time and space, communication within the group is more complex than in face-to-face interaction [9], becoming itself a task to be fulfilled that impedes users from their main duties.

The communication overhead can be reduced by the use of an adequate awareness mechanism. Most existing awareness systems were designed for real-time such as [11] or asynchronous such as [21, 18]. However, none of these mechanisms provide the possibility of both working in isolation and offering suitable information that prevents blind modifications.

In this paper, we propose a new interaction work mode for avoiding blind modifications while allowing users to work asynchronously in isolation. This mode provides information in real-time about group members activities. Changes are extracted from activities performed by users working in isolation. Then, details of changes could be filtered according to user preferences in order to preserve their privacy. The information is sent within the group for computing awareness information that is then displayed to users. Depending on details of filtered changes, users will be more or less aware of concurrent changes, and in this way, prevented from performing blind modifications in the document. For example, if a user chooses a low privacy level, all details of his modifications are sent to other users. Therefore, these concurrent modifications can be precisely localised and displayed to other users. Consequently, these users are aware of others' modifications and can avoid working in concurrently modified areas of the document. On the contrary, if a user chooses a high privacy level for his modifications, the only provided information is the name of the document currently modified by that user.

The main assumption of our novel writing mode is that users are connected most of the time, even when they work in isolation. This assumption seems to be realistic 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. – Furthermore, network ubiquity will continuously expand in the near future. It is worth to point out that disconnected work is still supported but with the risk of performing blind modifications.

Our paper is organized as follows. The section 2 motivates our approach and illustrates problems regarding blind modifications while working in isolation. Section 3 describes Gasper, our approach that addresses the issues of blind modifications during collaborative work by introducing the concept of *ghost operations*. In section 4 our main motivating example is revisited to show how blind modifications could be prevented. Section 5 compares Gasper with related approaches. The last section presents some concluding remarks and directions of future work.

2. MOTIVATING EXAMPLES

In this section we are going to present collaborative scenarios illustrating some of the problems regarding blind modifications while working in isolation. We focus on one example in the domain of software engineering, but we also present scenarios regarding collaborative writing of research papers and wiki pages.

Scenario 1 - Software Engineering

Consider the scenario involving three software engineers collaborating on the source code of the same project as summarised in the table 1.

Although at the beginning they divide their work according

Step	Actions of developer 1	Actions of developer 2	Actions of developer 3
1	op_1: removes method isReal() from class Integer	op_2: updates method isReal() from class Integer	op_3: creates test class IntegerTest to check methods of class Integer
2	COMMIT		
3		UPDATE *(a conflict is detected between operations op_1 and op_2)*	UPDATE *(the test class TestInteger does not compile)*
4		solves conflict between op_1 and op_2 by re-inserting new method isReal()	removes test for method isReal()
5		COMMIT	
6			UPDATE & COMMIT *(no test for the method isReal())*
7	UPDATE	UPDATE	

Table 1: Summary of scenario 1.

to predefined tasks, their modifications will overlap later on during their isolated work since their tasks involve some common classes.

In step 1 of the scenario, the first developer decides to remove the method isReal() from the class Integer illustrated in figure 1. He therefore performs operation op_1. His decision is motivated by the fact that he thinks that this method is not used throughout the project.

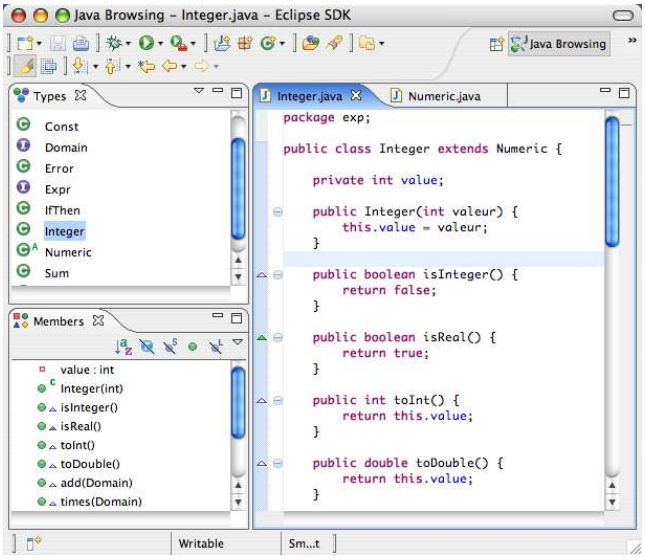

Figure 1: Initial document state

Concurrently, the second user that uses the functionality of class Integer realises that the method isReal() should be corrected - it should return false as an integer should not be considered to be a real. Therefore he performs operation op_2.

In the same time with the modifications of the two users, the third user tests the class Integer by creating the test class IntegerTest. One of the added methods in that class is the test method for isReal() as represented by operation op_3.

In step 2, the first user commits his changes. In step 3, both developers 2 and 3 update their copies of the project. At this moment, developer 2 is informed about the conflict between the removal of method isReal() committed by user 1 and his update of that method. After updating his copy, user 3 notices that his test class IntegerTest does not compile anymore since the tested method isReal() was removed by user 1.

In step 4, developer 2 solves the conflict between operations op_1 and op_2 by re-inserting the method isReal() that he modified since he needs it for his task. Concurrently, the third developer removes the test method testIsReal() since the tested method was removed.

In step 5, developer 2 commits his changes. In step 6, the third developer updates his copy. Since he is not aware of any conflict nor any compilation errors, he decides to commit his changes. He does not notice that the method isReal() was re-inserted in order to write a test for this method.

Afterwards, in step 7, both developers 1 and 2 update their copies. At the end of this scenario the copies of the shared project converge, but the method isReal() remains untested.

Due to blind modifications performed by users while working in isolation, the following undesired situations occurred:

- User 1 deleted the method isReal() which was finally re-inserted by user 2. His work was useless and produced side-effects for the tasks of other users.
- User 2 modified the method isReal() but due to its removal by user 1 he needed to re-perform his initial change.
- User 3 wrote the test for method isReal() and was obliged to remove it. Therefore, he performed "useless" work and finally he did not realise that his task is incompleted.

Scenario 2 - Collaborative Authoring of Wiki Pages

A similar example could be described in the context of collaboration over a wiki system. Suppose two users are working on the same article on Wikipedia. The first user starts to edit the content of the page in order to correct misspellings and grammar errors. Concurrently, the second user decides to remove a section from the article and commits his changes. The first user is not aware that a new version of the page was published and continue to correct misspellings even in the section that was removed in the published version of the page. This scenario illustrates that blind modifications may lead to useless work.

Scenario 3 - Collaborative Authoring of Research Papers

Similarly, consider two students concurrently writing a research paper. During the reviewing phase they concurrently decide to illustrate the same definition by means of an example. Their added example is based on the same main ideas, but formulated differently. After merging their changes, the definition will be illustrated by the same basic example described two times. The collaborative work would have been more productive if they would have been aware about the activity of the other user and decide to compose together the example. This scenario shows that blind modifications might cause redundant work where the same initially unknown task is performed twice.

As we have seen in this section *blind modifications* might occur during work in isolation and may produce undesired effects such as *useless and redundant works*. Compensation of such effects might considerably diminish the gain of concurrent work. Therefore, we see the need of a notification mechanism that informs in real-time users about concurrent activities performed in isolation.

3. PREVENTING BLIND MODIFICATIONS BY MEANS OF GHOST OPERATIONS

In this section, we present our approach called Gasper for preventing blind modifications in asynchronous collaborative writing that might occur while working in isolation.

3.1 Standard Model for Asynchronous Collaboration

In what follows we present an initial model for asynchronous collaborative editing that will be refined throughout this section in order to define our new mode of interaction. For the sake of simplicity, we restrain our study to asynchronous collaborative editing systems relying on a central server. However, our proposed approach could be adapted to fully decentralized collaborative editing systems.

An asynchronous collaborative editing system is composed of n user sites and a server acting as a central repository. Each user associated with a site works on his copy of shared documents. A user can perform the following actions:

- modifies a document by generating operations that are immediately applied on his local copy,
- makes available his local changes to other users by committing his local operations to the repository. The user is not allowed to commit his local operations if his local copy is not up-to-date, i.e. if some non-integrated remote operations are available on the repository.
- updates his local copy of a document by integrating remote operations from the repository.

Operations can model any changes targeting a document such as insertion, deletion and update of lines in that document or concerning a document file system such as moving a file from a directory to another one. In order to model any type of operation we defined an operation by using the following grammar.

operation = <*type, (parameter)**>
parameter = *(pname, ptype, pvalue)*
type = INSERT | DELETE | . . . | UPDATE | MOVE
ptype = STRING | INTEGER | REAL
pname = ([A .. Z] | [0 .. 9])+
pvalue = ([A .. Z] | [0 .. 9])+

An operation is composed by a type and a list of parameters. The type of an operation can be *INSERT*, *DELETE*, *UPDATE*, *MOVE*, etc. Each parameter is a tuple composed by the name of the parameter, its type and its value. The type of a parameter can be string, integer or real. The name and value of a parameter can be any sequence of characters and numbers. We suppose that a parser can check that the value of a parameter corresponds to its type.

For instance, an operation of insertion of line 4 with the content *"Preventing blind modifications"* inside the document file.txt generated by $User_2$ will have the form: op_2=<*insert, [(initiator,integer,2), (file,string,"file.txt"),(line,integer,4), (content, string,"Preventing blind modifications")]*>.

An operation executed by $User_3$ of moving the document called paper.tex from the directory /opt/doc/draft to the directory /opt/doc/final will be represented by the form op_3= <*move, [(file,string,"paper.tex"),(fromDirectory,string, "/opt/doc/draft"), (toDirectory,string,"/opt/doc/final")]*>.

For the sake of simplicity, throughout this paper we denote operations by specifying their type, target and content ignoring other parameters. For instance, the operation *op=insert(file.txt,4,"Preventing blind modifications")* denotes the insertion of line 4 into the file called file.txt, the content of the inserted text being *"Preventing blind modifications"*.

After its generation, a local operation can be committed to the repository or it can be aborted. Therefore, a local operation can be in any of the following three states as shown in the figure 2: LocallyExecuted, Committed and Aborted.

The model of asynchronous collaboration presented in this

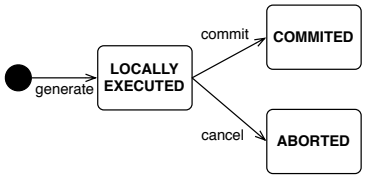

Figure 2: State diagram of an operation in asynchronous collaboration.

section is known as the *Copy-Modify-Merge* (CMM) paradigm [5]. The CMM work mode designed several decades ago requires that users connect to the server only for committing and updating their local copies and does not assume that users remain connected during the collaboration.

3.2 Enhancing Asynchronous Collaboration with Ghost Operations

Nowadays collaborators are connected most of the time. In this context, connectivity can be exploited to enhance the CMM paradigm. We saw in the previous section that working in isolation might generate blind modifications. Therefore, our goal is to extend the CMM paradigm in order to prevent blind modifications.

Under the assumption that a user is continuously connected, it is possible that she receives in real-time non-committed parallel modifications in order to annotate his local copy of the document. This presumes that users agree to send in real-time their local non-committed operations. Unfortunately, this assumption might violate user privacy as users may not agree to send draft changes of their work.

Privacy can be controlled by groups and individuals. Privacy issues can be resolved by using an access control or content control strategy [6]. Access control solutions restrict access of unauthorized users to data. Content control solutions remove confidential information such that the filtered information can be shared without violating user privacy. In our approach we used content control strategy for dealing with privacy related to changes. There is a trade-off between privacy and the usefulness of awareness: if users agree to have less privacy, other group members are provided with rich awareness.

We therefore filter non-committed local operations before sending them to other users by masking some operation parameters. We call these operations *ghost operations*. For example, suppose two users edit collaboratively the wiki page about the Star Trek Ferengi Rules of Acquisition:

#1	Once you have their money, never give it back
#242	More is good, All is better

Suppose the first user inserts as the second line the 19th rule "#19 Satisfaction is not guaranteed" by performing the operation op_1=*insert(FerengiRulePage,2,"#19 Satisfaction is not guaranteed")* in order to obtain the document:

#1	Once you have their money, never give it back
#19	Satisfaction is not guaranteed
#242	More is good, All is better

The user can decide to filter his operation and generate the ghost operation $g(op_1)$=*insert(2,34)* and send it in real-time to other sites. The ghost operation masks the content of the inserted line and replaces it with its length, i.e. 34.

When ghost operations are received at remote sites, awareness information concerning group activity can be computed. One form of representing awareness provided by ghost operations is document annotation. For instance, in our example, the first user might be presented with the following version of the document, where the annotation mark ⋆⋆ informs about concurrent non-committed changes referring to that location of the document.

	#1	Once you have their money, never give it back
⋆⋆	#242	More is good, All is better

A *ghost operation* is the result operation obtained by filtering an original operation according to user privacy preferences. In the rest of the paper, in order to distinguish between original form and ghost form of an operation, we will refer to them as *real operation* and *ghost operation* respectively. More formally a ghost form of an operation *op* is defined by using the following grammar :

g(operation)	= <*filter(type), (filter(parameter))**>
filter(type)	= type | EDIT
filter(parameter)	= NULL | *(pname, ptype, pvalue)* | *(filteredPName, filteredPType, filteredPValue)*

The type of a real operation might be masked in the corresponding ghost operation. The list of parameters of a real operation might be masked in the corresponding ghost operation. The ghost operation might not contain a parameter belonging to the real operation, it might contain it in the original form or it might filter it. A filtered parameter is formed by the filtered name, the filtered type and the filtered value of the original parameter.

For instance, for the original operation of insertion of line 4 with the content *"Preventing blind modifications"* inside the document file.txt generated by $User_2$ op_2=<*insert, [(initiator,integer,2),(file,string,"file.txt"),(line,integer,4), (content, string,"Preventing blind modifications")]*>, the following ghost operations might be generated:

- $g(op_2)$=<*insert, [(initiator,integer,2), (file,string, "file.txt"),(line,integer,4),(contentSize, integer, 30)]*>. In this case the ghost operation masks the content of the inserted line, by specifying the file name and the line where the insertion takes place as well as the size of the inserted content.
- $g(op_2)$=<*edit, [(file,string,"file.txt"),(line,integer,4)]*>. In this case the ghost operation masks the identity of

the user that generated the operation, the type of the operation and the content of the inserted line. It just indicates that a modification has been performed by a certain user in the document file.txt at line 4.

In the rest of the paper we are going to use a simplified form for representing ghost operations in the same way we represent real operations.

In what follows we enumerate some of the important characteristics of a ghost operation.

Ghost operations are not designed to be integrated in the local copy of the document, but rather "to annotate" the document. Depending on the carried data, various annotation forms can be computed. If the carried data are sufficient for locating the concurrent changes, annotations can form an overlay model that is presented to users over their document view. If the granularity of the provided localisation information is the document, then concurrent activities can be depicted over shared documents but not within them. A visual representation of this awareness information was proposed in [14]. Even if all the parameters of the real operation were filtered in the ghost operations, it is possible to count the number of concurrent operations performed by group members. On the contrary, if the form of ghost operation equals the real operation, i.e. no information is filtered, future committed changes and their consequences can be predicted and previewed in real-time.

Generally a community of users is structurally organised into groups and subgroups. According to his privacy preferences, a user can assign different privacy rules to groups and individuals of his community. Therefore, a real operation is filtered according to various privacy rules and might generate several ghost operations that are sent to different members of the group. The ghost operations can be directly sent to group members without the need to pass through the central repository.

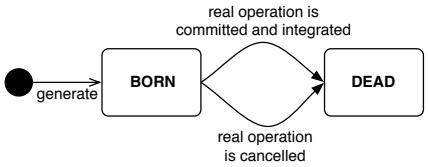

Figure 4: State diagram of a ghost operation in Gasper.

In our approach, the lifecycle of a ghost operation can be summarised as follows. After its reception at a site a ghost operation is in the state BORN. It remains in that state until the real operation is committed and integrated on that site when it passes into the state DEAD and disappears from the system. Due to the abortion of the corresponding real operation, this state is also reached by a ghost operation. This case happens when a user decides to cancel his local changes and do not commit the real operation.

Details of our approach presented in this section are summarised by the architecture of Gasper illustrated in figure 3 and presented in what follows.

A user site is composed of the following components:

- the *document model* and several *annotation models*
- the logs of real and ghost operations
- the filtering module
- the broadcaster and the receiver

Operations describing user changes are kept in logs. Each local site maintains a log with the locally executed real operations and a log with real committed operations. As ghost operations do not affect the state of a document, they are kept in separate logs. Therefore, each site contains a local ghost operation log and a remote ghost operation log.

Changes performed on the document model generate a list of real operations that are kept in the *local real operation log*. According to his privacy preferences, each user maintains for every set of changes that he performed a list of privacy rules associated to various groups and individuals of his community. Each real operation is filtered according to the corresponding privacy rules and generates the corresponding ghost operations that are kept in the *local ghost operation log*. Operations in the *local ghost operation log* as well as operations in the *local real operation log* that need to be committed are sent to the *broadcaster*. The broadcaster sends afterwards the ghost operations to the corresponding users and the real operations to the repository.

When the *receiver* receives the remote real operations, these operations are kept in the *real committed operation log* and executed on the *document model*. When the *receiver* receives the ghost remote operations, these operations are kept in the *remote ghost operation log* and they are executed to modify the *annotation models*.

4. REVISITING MOTIVATING EXAMPLE

In this section we revisit the motivating example in the domain of software engineering presented in section 2 by showing how blind modifications can be prevented by applying our approach.

Consider that the three developers have performed their actions described in step 1 of table 1. The first user removes method isReal() by generating the operation op_1=*delete(User*$_1$*,Integer.java,15-18)*. This operation removes the lines 15 to 18 describing the definition of the method isReal() from the file Integer.java. The second user modifies method isReal() by updating the content of the line 16 with the content return false;. Therefore, operation op_2=*update(User*$_2$*, Integer.java,16,"return false;")* is generated. The third user creates the file TestInteger.java and inserts the test methods for the class Integer. For the sake of simplicity, we do not define the form of the generated operations. In the what follows, these operations will be referred as operation op_3.

Further, suppose that users decide to send ghost operations describing their activity while working in isolation.

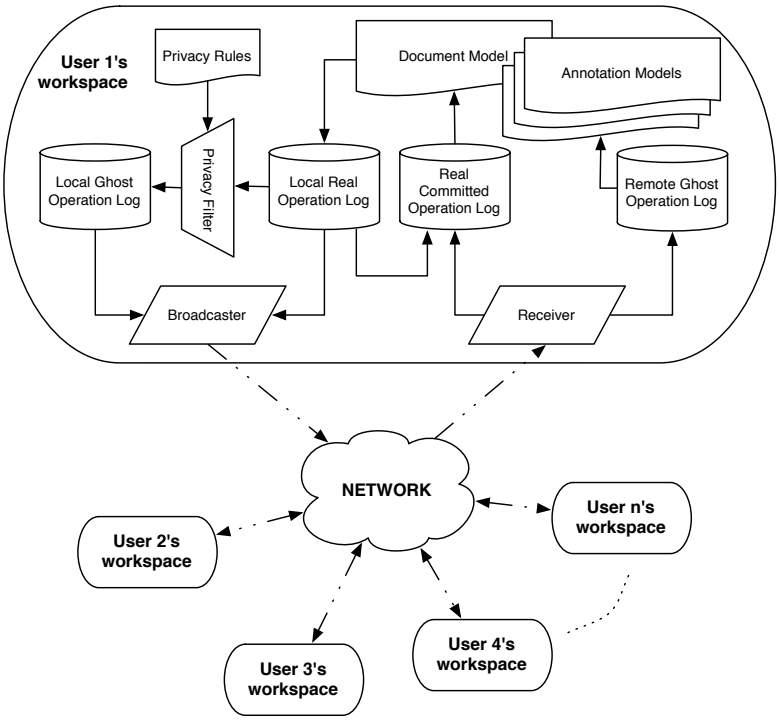

Figure 3: Gasper Architecture

Generated operations	Privacy filter	Shared ghost operations
op_1=*delete(User$_1$,Integer.java,15-18)*	do not filter	$g(op_1)$=*delete(User$_1$,Integer.java,15-18)*
op_2=*update(User$_2$,Integer.java,16,"return false;")*	filter content	$g(op_2)$=*update(User$_2$,Integer.java,16,-)*
op_3	do not send ghost	-

Table 2: Summary of operations.

The first user decides to apply the privacy policy allowing to send the full content of his modifications as ghost operations. Therefore, he sends the following ghost operation: $g(op_1)$=*delete(User$_1$,Integer.java,15-18)*.

In order to make other users aware about his modification, the second user decides to apply the privacy policy that hides the content of his changes but shares their location. Therefore, the form of the generated ghost operation is $g(op_2)$=*update(User$_2$,Integer.java,16)* signifying that line 16 is under modification.

The third user performing testing decides not to send any ghost operations regarding the changes he performs by applying the strongest privacy policy.

The real and their corresponding ghost operations are summarised in table 2.

In the following we show how ghost operations can make users aware about concurrent changes and avoid undesired operations. Let us analyse what happens at the site of the first user who deletes the method isReal(). After the reception of the ghost operation sent by the second user $g(op_2)$, awareness information concerning activity of the second user can be presented as depicted in the figure 5. Since the ghost operation $g(op_2)$ contains information about the target file, it is possible to indicate by means of a marker 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. The position of the modified line is computed by using the line number indicated by the ghost operation.

An annotation can be associated with a marker in order to provide additional information regarding concurrent changes. Assume the editor is capable of finding the methods associated with a certain line range. In this way, the user can be informed that there is a conflict between his local change and the remote ones. For instance, the associated annotation in figure 5 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.

Let us analyse what happens at the site of the second user. After modifying the method isReal(), the ghost operation of the first user arrives at the site and awareness information will be displayed as shown in Figure 6. In the right hand side

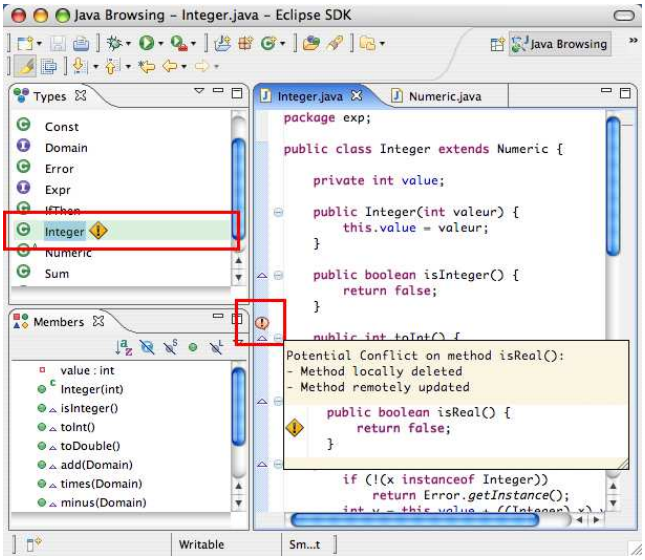

Figure 5: Interface at the first site after integration of ghost operations

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.

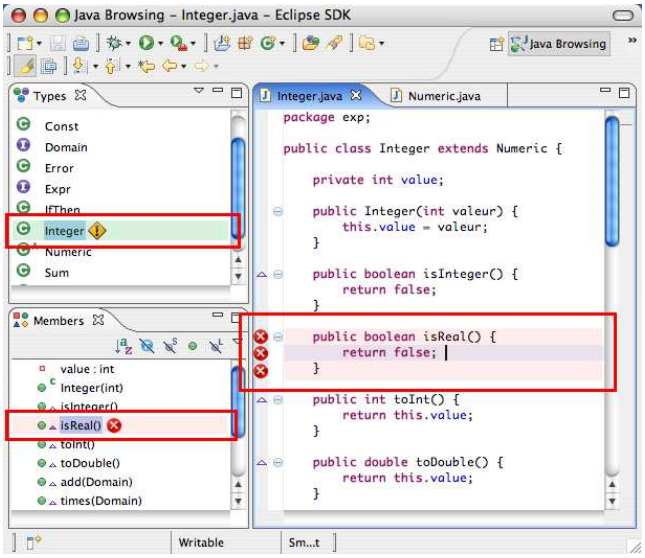

Figure 6: Interface at the second site after integration of ghost operations

At the site of the third user, after the ghost operations $g(op_2)$ and $g(op_1)$ arrive, awareness information regarding modification of class Integer is presented as shown in the top left hand side window of the interface shown in the Figure 7. In this way, the user could examine the concurrent modifications performed in class Integer and be presented with almost the same view as the second user. The third user could also initiate a communication with the other users that performed concurrent changes.

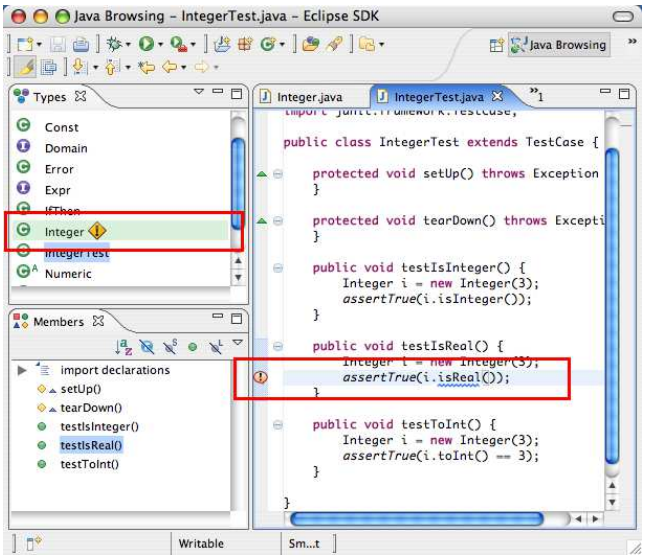

Figure 7: Interface at the third site after integration of ghost operations

Contrary to the initial scenario described in section 2, the undesired situations produced by blind modifications do not occur:

- User 1 will not validate his removal of the method isReal() as he is informed that another user is currently modifying it.
- User 2 will notice that another user wants to remove the method he is currently modifying . He can initiate a communication with that user.
- User 3 is informed that the class Integer is currently modified by two users and therefore decide to postpone testing this class at a later time.

As shown in this section, undesired effects of isolated work such as useless or redundant work can be avoided by providing awareness information in real-time.

5. RELATED WORK

Many approaches in the literature were dedicated to providing various awareness mechanisms while working in a collaborative environment. But most of these awareness approaches were developed for the communication on real-time in order to help users to coordinate their group work. For instance, multi-user scrollbars represent the relative location of each user in a large document by means of a coloured bar layered beside the conventional scrollbar [3], telepointers indicate where users are pointing [23] and radar views [12] display miniatures of user workspaces which might contain user pointers. However, these approaches cannot be used for avoiding blind modifications in asynchronous communication.

Few awareness mechanisms were proposed for the asynchronous mode. In the field of configuration management systems, the oldest mechanism for avoiding blind modifications is the CVS watches [5]. Watches allow developers to specify the artifacts they want to monitor. When a developer wants to change an artifact, he announces his intent of modification by invoking a certain command. This command triggers notifications by means of emails to developers that registered for the change of those artifacts. However, this approach offers limited awareness information by means of emails and does not provide a presentation mechanism.

Most awareness approaches for the asynchronous communication concentrated mainly on change awareness. These approaches highlight changes made by other participants over time to an artifact such as a document or workspace. An initial framework on change awareness was proposed in [10] and then refined in [21]. These approaches maintain a user aware about changes that were performed and published while he was working in isolation. They do not present changes that are concurrently performed and not yet published and therefore, these approaches do not prevent blind modifications.

In [17] the authors proposed an editing profile that counts operations performed by users on different parts of the document, such as paragraphs, sentences and words in the case of textual documents. The editing profile provides an awareness mechanism regarding the hot areas of the document with the highest number of changes. Unfortunately, it is computed only after users perform an update of their copy of the document. Therefore this mechanism is dedicated for change awareness and does not help avoiding blind modifications. The editing profiles proposed in [17] can be computed in our approach by using the committed operations. Additionally, in our approach the same profile can be computed in real-time before changes are published. In this way blind modifications might be prevented.

The State Treemap [14] is an awareness widget that offers support for the multi-synchronous collaboration. It is designed to inform users about states of the shared documents. Different states are defined for a document such as LocallyModified, PotentiallyConflict – when two copies of the document are modified and none of the changes are published yet – or WillConflict – when a document copy is modified locally and some changes on that document have been committed. Since the computation of the awareness information is based on the states of document copies, the approach has few limitations. Firstly, the granularity of the awareness information is the document and therefore it is impossible to locate concurrent modifications within the document. Secondly, it provides only qualitative information as users are informed about divergence between two copies of the same document, but it does not provide quantitative information as no measure for the divergence between two copies is provided.

Palantìr [18] provides awareness information about concurrent modifications performed in isolation in the context of configuration management systems. It is based on the same principle as State Treemap, the main difference being that a severity information that computes the amount of changes performed among documents was added. Unfortunately, the granularity of provided information is still the document. Moreover, the severity metrics does not provide enough information to infer changes that could cause potential conflicts at the merging phase. For instance, suppose a document was concurrently changed by two users with a severity of 20% and 30% respectively. When the two versions of the document are merged, no metric is provided for the computation of the severity of the merged documents as there is no information whether concurrent changes overlap or not.

Concerning quantitative measurement of divergence between document copies, the approach proposed in [15] provides divergence metrics. Contrary to Palantìr and State Treemap approaches, metrics are computed using operations modeling concurrent changes and not regarding events triggered by document state transitions. Merging of these concurrent operations is simulated in real-time on each site making possible the computation of various metrics. For instance, it is possible to compute the amount of changes performed on each document as in Palantìr, but also an amount of conflicting/overlapping changes. However this approach provides only metrics for each shared document but does not localise changes within documents.

Gasper is localising exactly the changes in a document and additionally deals with ghost operations that maintain user privacy while providing group awareness. Gasper can be seen as a more general approach than Palantìr, State Treemap and divergence metrics approaches in the sense that Gasper can simulate the functionality of these three approaches. If ghost operations carry data about location of the targeted artifacts, it is possible to compute the information requested by State Treemap approach. If additionally the amount of changes is included in ghost operations, then severity measure proposed in Palantìr can be included. Furthermore, if ghost operations contain precise information about the location and size of changes within the documents, then the divergence metrics proposed in [15] can be evaluated.

The NICE [19] approach provides a notification mechanism for both real-time and asynchronous collaboration. Various notification policies can be defined such as system-triggered (instant or scheduled) or user controlled. If appropriate settings for the notification policies are set, the system could be used to avoid blind modifications. However, a user can be aware of other user changes only when he decides to integrate them with his own as reception of operations at one site implies their immediate integration. The approach does not deal with ghost operations that offer support for maintaining user privacy and the possibility of being aware about group changes without the integration of these changes.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a novel collaborative writing mode for avoiding *blind modifications* that occur during isolated work. We illustrated by means of scenarios the undesired consequences of blind modifications such as redundant or useless works. Based on the assumption that users are most of the time connected including work in isolation, we proposed to exploit their connectivity by continuously providing them with awareness information about group activity. We introduced the concept of *ghost operations* that carry information about performed operations

while preserving user privacy preferences. As an example, we showed how awareness information provided by ghost operations could be represented in a software engineering development environment. We expect that provided awareness information will generate group communication and autocoordination between users in order to prevent conflicts.

We are currently refining the implementation of the approach proposed in this paper. We plan to investigate the usability and the benefits of our approach by performing user studies. Another direction of future work is to define new metrics that can be computed with our approach and propose novel visualisation interfaces.

7. REFERENCES

[1] Subethaedit. *Collaborative text editing. Share and Enjoy*, 2007. http://www.codingmonkeys.de/subethaedit/.

[2] Subversion. *Next-Generation Open Source Version Control*, 2007. http://subversion.tigris.org/.

[3] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby. The User-centered Iterative Design of Collaborative Writing Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI'93*, pages 399–405, Amsterdam, Netherlands, 1993. ACM Press.

[4] E. Beck and V. Bellotti. Informed Opportunism as Strategy: Supporting Coordination in Distributed Collaborative Writing. In *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW'93*, pages 233–246, Milano, Italy, September 1993. Kluwer Academic Publishers.

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

[6] M. Boyle and S. Greenberg. The Language of Privacy: Learning from Video Media Space Analysis and Design. *ACM Transactions on Computer-Human Interaction*, 12(2):328–370, 2005.

[7] L. Ede and A. Lunsford. *Singular Text/Plural Authors: Perspectives on Collaborative Writing*. Southern Illinois University, 1990.

[8] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):39–58, January 1991.

[9] J. Galegher and R. E. Kraut. Computer-mediated Communication for Intellectual Teamwork: A Field Experiment in Group Writing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 1990*, pages 65–78, Los Angeles, California, USA, October 1990. ACM Press.

[10] C. Gutwin. *Workspace Awareness in Real-time Groupware Environments*. Ph.D. Thesis, Department of Computer Science, University of Calgary, Calgary, Canada, 1997.

[11] C. Gutwin and S. Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work - JCSCW*, 11(3):411–446, September 2002.

[12] C. Gutwin, S. Greenberg, and M. Roseman. Workspace Awareness Support with Radar Views. In *Conference Companion on Human Factors in Computing Systems - CHI'96*, pages 210–211. ACM Press, April 1996.

[13] P. B. Lowry, A. Curtis, and M. R. Lowry. Building a Taxonomy and Nomenclature of Collaborative Writing to Improve Interdisciplinary Research and Pratice. *Journal of Business Communication*, 41(1):66–99, January 2004.

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

[15] 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. Society for Design and Process Science.

[16] S. Noël and J.-M. Robert. Empirical study on collaborative writing: What do co-authors do, use, and like? *Computer Supported Cooperative Work - JCSCW*, 13(1):63–89, March 2004.

[17] S. Papadopoulou, C.-L. Ignat, G. Oster, and M. Norrie. Increasing Awareness in Collaborative Authoring through Edit Profiling. In *Proceedings of the IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2006*, pages 1–10, Atlanta, Georgia, USA, November 2006. IEEE Computer Society.

[18] 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. IEEE Computer Society.

[19] H. Shen and C. Sun. Flexible Notification for Collaborative Systems. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW'02*, pages 77–86, New Orleans, Louisiana, USA, November 2002. ACM Press.

[20] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent Adaptation of Single-user Applications for Multi-user Real-time Collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, December 2006.

[21] J. Tam and S. Greenberg. A Framework for Asynchronous Change Awareness in Collaborative Documents and Workspaces. *International Journal of Human-Computer Studies - IJHCS*, 64(7):583–598, July 2006.

[22] S. G. Tammaro, J. N. Mosier, N. C. Goodwin, and G. Spitz. Collaborative Writing Is Hard to Support: A Field Study of Collaborative Writing. *Computer-Supported Cooperative Work - JCSCW*, 6(1):19–51, March 1997.

[23] S. Xia, D. Sun, C. Sun, and D. Chen. Collaborative object grouping in graphics editing systems. In *Proceedings of IEEE 2005 International Conference in Collaborative Computing (CollaborateCom'05)*, San Jose, California, USA, December 2005.