Undo in Peer-to-peer Semantic Wikis

Charbel Rahhal, Stéphane Weiss, Hala Skaf-Molli, Pascal Urso, Pascal Molli

Undo in Peer-to-peer Semantic Wikis

Charbel Rahhal, Stéphane Weiss, Hala Skaf-Molli, Pascal Urso, and Pascal Molli

LORIA – INRIA Nancy-Grand Est, Nancy Université, France
{Charbel.Rahal, weiss, skaf, urso, molli}@loria.fr

Abstract. The undo mechanism is an essential feature in collaborative editing systems. Most popular semantic wikis support a revert feature, some provide an undo feature to remove any modification at any time. However this undo feature does not always succeed. Supporting the undo mechanism for P2P semantic wikis has never been tackled. In this paper, we present an undo approach for Swooki the first P2P semantic wiki. We identify the problems to resolve in order to achieve such mechanism for P2P semantic wikis. We give the definition of the undo and the properties that must ensure. This approach allows both a revert and a permanent successful undo.

1 Introduction

Wiki systems are the most popular collaborative editing systems. Thanks to a simple syntax, users can build complex text documents, including tables, pictures or videos. As collaborative editors, wiki systems provide an undo mechanism. This mechanism is integrated in two forms, either through a *revert* which allows to return to an old version, or through an *undo* allowing to remove any modification from the current version [24].

In spite of their success, wiki systems have some limitations such as:

Centralization Most of existing wikis are centralized: this implies a high cost to ensure scalability, censorship issues, data availability and durability issues especially in case of failure;

Low structuring Wiki systems are low-structured, they suffer in the navigation and the search, i.e. it is hard to navigate and to find relevant information in wikis [6]. Wiki content is only human readable and it is not accessible and readable by machines, hence, it cannot be reused in external applications.

To overcome these limitations, two orthogonal solutions are proposed : P2P wiki systems and semantic wiki systems.

P2P wiki systems [27, 12] are based on a decentralized architecture and optimistic replication[19] mechanism to improve scalability and data availability. As traditional wiki systems, they suffer from low structuring.

Semantic wiki systems [26, 20, 6] allow users to incorporate some computer readable information in wiki pages. Such information can be used to improve navigation and search. However, they suffer from centralization limitations.

Swooki [21] aims at conciliating both directions, it combines the advantages of P2P systems and semantic wikis. Swooki is a semantic wiki inspired from Semantic MediaWiki [26]. Moreover, Swooki supports massive collaboration, fault tolerance, off-line work mode and ad hoc collaboration thanks to its P2P structure and to the replication of semantic wiki pages on different sites. Unfortunately, this approach does not offer any undo mechanism.

In the literature, several collaborative editing systems offer an undo mechanism. Existing semantic wikis are centralized, therefore their undo mechanism is not adequate for the P2P environment. On the contrary, some undo frameworks were devised for distributed collaborative systems, however they do not support semantic wiki data type. The data type maintained in semantic wikis is the wiki pages and the semantic annotations storage. Our goal is to design an undo mechanism that is compatible with P2P constraints, that supports the semantic wiki data type and that ensures the consistency between the wiki pages and the semantic storage.

In this paper, we propose an undo mechanism for Swooki. We define the property that this mechanism must ensure. This mechanism supports both undo and revert features. We develop the undo component and the needed algorithms and extensions to instantiate this undo mechanism in Swooki.

The paper is organized as follows. In section 2, we motivate the need for the undo mechanism. Section 3 presents existing approaches for the integration of the undo mechanism in collaborative editing systems. Section 4 presents Swooki. An overview of the undo in Swooki approach is given in section 5. Section 6 describes the implementation of the approach. The last section concludes the paper.

2 Motivation

Similarly to classical collaborative editors, P2P semantic wikis require supporting the undo feature for many reasons:

– Undo is a user required feature. Indeed, users can use the undo feature as a powerful way to recover from their proper errors.
– In collaborative editors, when two or more users modify the same data, the system proposes a document based on all modifications. This merge algorithm is a best-effort and is not able to produce the result expected by users. The undo feature is useful to recover from unexpected result.
– We consider a P2P semantic wiki as an open system where anyone can join and leave. Since anyone can join, malicious users can also join. As a result, the undo feature can be used to remove the impact of vandalism acts.

In all these cases, the expected result matches the undo definition [23]:

"Undoing a modification makes the system return to the state it would have reached if such modification was never produced."

To achieve such a goal, the revert feature seems to be adequate: we can remove the whole content and add a previously created one. Unfortunately, the revert feature does not allow to undo any modification.

Indeed, concurrent operations can be received in different orders. For instance, starting from an initial version V_0 (see Figure 1). A $user1$ on Site1 inserts a second line, though he generates a new version V_1 including the M_1 modification. Another $user2$ on Site2 inserts concurrently a second line, he generates a new version V_2 including the M_2 modification. Unfortunately, he made a mistake about the value of the car top speed. The integration of the M_1 and M_2 modifications on both sites results into a version V_3 including both of them. In order to correct the mistake, each site has different alternatives. Site1, which has integrated first M_1, can restore the versions V_1 or V_0. V_1 is the expected result since it contains all the modifications except the erroneous one. On the contrary, Site2 which has received first M_2, can return to V_0 or V_2. V_2 contains the erroneous modification while V_0 does not contain M_1. As a result, Site2 is not able to cancel only the erroneous modification using a revert function without a lost of the Site1 modification.

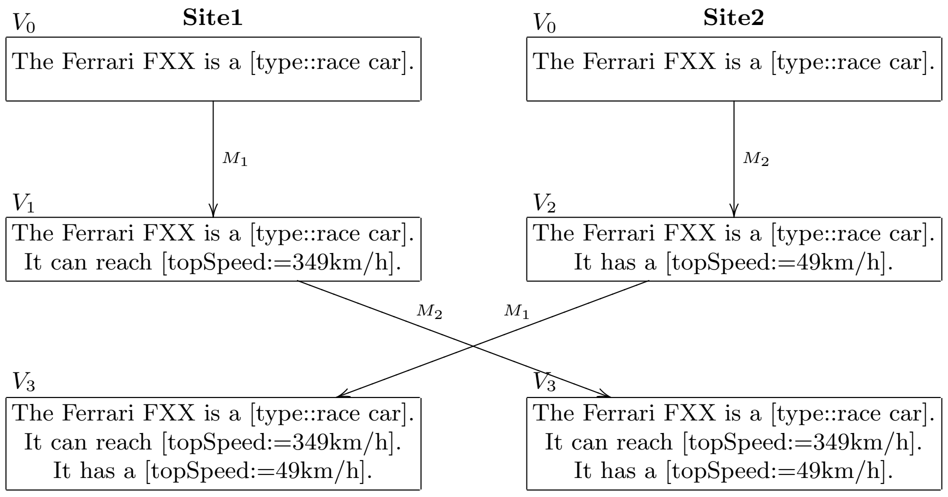

Site1
Site2
V_0
The Ferrari FXX is a [type::race car].
V_0
The Ferrari FXX is a [type::race car].
M_1
M_2
V_1
The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h].
V_2
The Ferrari FXX is a [type::race car]. It has a [topSpeed:=49km/h].
M_2
M_1
V_3
The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h]. It has a [topSpeed:=49km/h].
V_3
The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h]. It has a [topSpeed:=49km/h].

Fig. 1. Concurrent editing scenario

Another common idea is to undo changes by doing the inverse modification. Unfortunately, this does not achieve the undo definition. Figure 2 illustrates this case.

Starting from version V_3 the result of the previous example, a $user3$ on Site3 generates a malicious modification M_4 by deleting the whole document. M_4 deletes all the document lines. Concurrently, $user1$ on Site1 deletes the third line that contains the error. The inverse modification of M_4 inserts the three lines. As a consequence, when $user1$ on Site1 undoes M_4, it reinserts the three lines

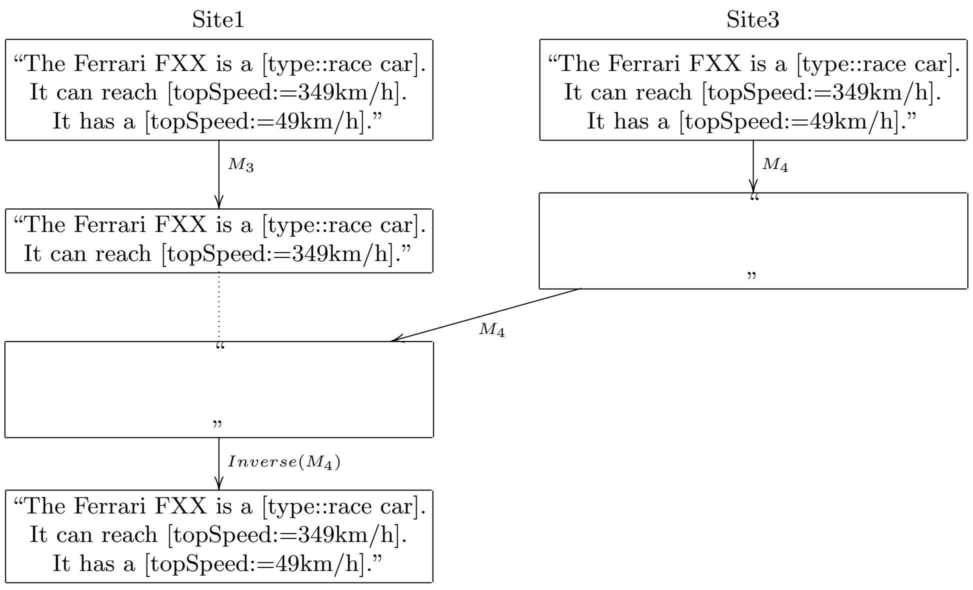

Site1
Site3
“The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h]. It has a [topSpeed:=49km/h].”
“The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h]. It has a [topSpeed:=49km/h].”
M_3
M_4
“The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h].”
“ ”
M_4
“ ”
$Inverse(M_4)$
“The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h]. It has a [topSpeed:=49km/h].”

Fig. 2. Undo using the inverse modification

and looses its proper modification M_3 which aims at deleting the third line. If M_4 was never produced, the document would be corrected to "The Ferrari FXX is a [type::race car]. It can reach [topSpeed:=349km/h].". A correct undo must return the document to this state. Our goal is to provide such an undo feature.

3 Related work

This section gives an overview about the undo mechanism in different collaborative editors.

3.1 Undo in semantic wikis

Semantic wikis such AceWiki [10], Rise [7] and WikiSar [4] do not support neither versioning for wiki pages nor versioning for metadata, hence they do not support an undo mechanism.

Makna [9] is a wiki based tool for distributed knowledge engineering. It extends the JSPWiki wiki engine with generic, easy to use ontology-driven components for collaboratively authoring, querying and browsing Semantic Web information. Makna supports versioning for pages and metadata within the pages, thus the revert feature is provided. However, it does not support the undo feature.

IkeWiki [20] is a semantic wiki with features to support collaborative knowledge engineering, different levels of formalization ranging from informal texts to formal ontologies, and it has a sophisticated, interactive user interface. IkeWiki supports also a revert feature to restore an old version. IkeWiki does not support a feature to undo modifications applied on a page version. In addition, the annotations about the wiki pages are outside the content of these pages. Tracking the annotations changes is not provided, an insert or a delete of annotations can not be detected.

SweetWiki [6] is a semantic wiki based on the CORESE engine. It supports WYSIWYG edition of pages and annotations, and use the CORESE engine and the SeWeSe library for all semantic operations : navigation, search and others. Pages are annotated using tags which are outside the content of the pages. SweetWiki does not support a versioning support for the formalized content, i.e. changes in the tags on pages is not tracked. SweetWiki supports a revert feature without an undo one.

Rhizome [22] supports a modified version of WikiML (ZML) that uses special formatting conventions to make semantic properties directly explicit in the page content. Pages are saved in RDF and another editor can be used to edit the RDF directly. Rhizome provides a native versioning of content and metadata. It provides only a revert feature without an undo one.

Semantic MediaWiki (SMW) [26] is an extension of MediaWiki that helps to search, organize, tag, browse, evaluate, and share the wiki content. SMW adds semantic annotations in the wiki pages in order to bring the power of the Semantic Web into the wiki. SMW inherits all the features of MediaWiki including revert and undo. While the revert always succeeds in restoring an old version, in some cases the undo can fail [1]. For instance, suppose that a modification M in a paragraph was followed by other modifications in the same paragraph. In this case, the undo of M can not work.

OntoWiki [3] and Powl [2] are web based applications designed to collaboratively build ontologies and create instances. Every change on any element such as knowledge model, concept, property or instance is logged. So they enable users to track, review and selectively roll-back changes. Consequently, they can offer both the revert and the undo features. Unfortunately, their undo mechanism is designed only for ontological elements and not for text.

3.2 Undo in different collaborative editors

Most of undo approaches were devised in the Operational Transformation [8] (OT) framework.

In [15], the authors propose to select which operation to undo. They also add the notion of conflict. If a conflict occurs, the undo is aborted. Therefore, this framework does not allow undoing any operation.

In [18], the authors propose a solution to undo operations in the inverse chronological order, i.e. from the last operation to the first one without skipping one. Therefore this approach does not allow undoing any operation.

[1] `http://en.wikipedia.org/wiki/Wikipedia:Undo#Undo`

The GOTO-ANYUNDO approach [23] is the first approach that allows any user to undo any operation at any time. This approach is designed for real-time editing and uses state vectors [11]. Since state vectors size is proportional to the number of site, this approach cannot be used in a P2P environment.

The COT approach [25] is an OT system designed for real-time editing which introduces the notion of "context vector". A context vector is associated to each operation and represents the operations executed before its generation. As state vectors, the cost of context vectors is compatible with real-time editing, however, they are not suitable in a P2P environment.

Distributed version control systems (DVCS) as Git [2] are P2P collaborative systems mainly used for source code editing. They compute a new patch to remove the effect of a previous one and treat it as a new patch. However, DVCS lack of a formal framework, indeed, there is no property to validate DVCS correctness.

The UNO[28] framework proposed an undo for P2P collaborative editing based on the Operational Transformation approach. The main idea of this approach is to devise new operation for counterbalancing previously made operations. This framework cannot be used directly to provide an undo feature in Swooki. However, we propose an undo mechanism inspired by this framework capable of undoing any modification at any time, i.e. supporting both a revert and an undo features.

4 Swooki System

Swooki [16, 21, 17] is the first P2P semantic wiki, it combines the advantages of P2P wikis and semantic wikis. Swooki is a P2P network of a set of autonomous semantic wiki servers (called also peers, nodes or sites) that can dynamically join and leave the network.

Every peer hosts a copy of all wiki pages where these pages embed semantic data and an RDF store. Every peer can autonomously offer all the services of a semantic wiki server. Swooki supports massive collaboration, improves data availability and has a high performance thanks to its total replication of shared data. It allows to query and access data locally without any data transfer. In addition, it enables off-line works and transactional changes.

As in any wiki system, the basic element is a wiki page and every wiki page is assigned a unique identifier $PageID$, which is the name of the page. The name is set at the creation of the page. If several servers create concurrently pages with the same name, their content will be directly merged by the synchronization algorithm.

A semantic wiki page *Page* is an ordered sequence of semantic wiki lines. A semantic wiki line L is a four-tuple $<$ LineID, content, degree, visibility $>$ where *LineID* is a unique line identifier, *content* is a string representing text and the semantic data embedded in the line. *degree* is an integer used by the

[2] http://git.or.cz/

synchronization algorithm. *visibility* is a boolean representing whether the line is visible or not. Lines are not deleted physically, they are just invisible in the view of the semantic wiki page.

A Swooki peer is composed of the following components (see figure 3):

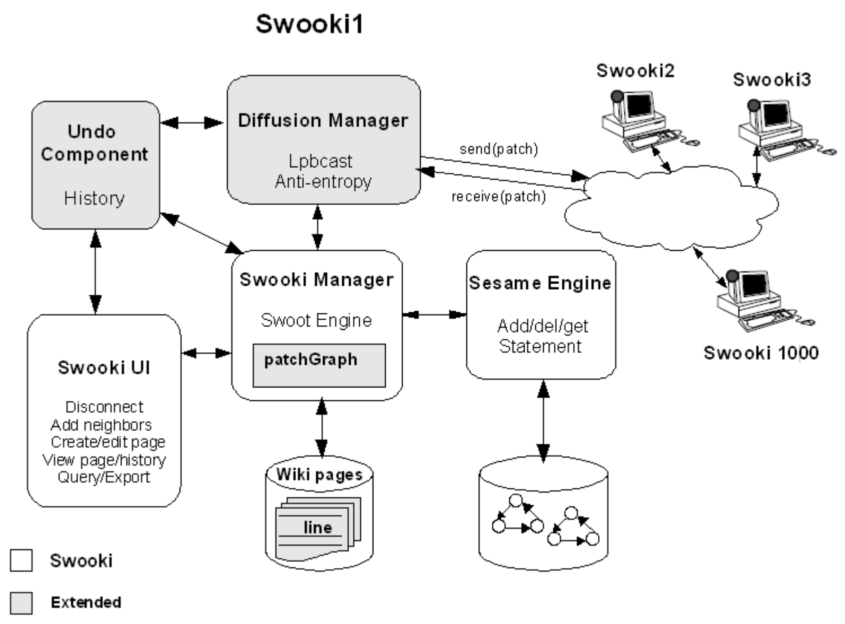


Fig. 3. Swooki architecture extended with the undo component

User Interface. The Swooki UI component is basically a regular wiki editor. It allows users to edit a view of a page by getting the page from the Swooki manager. Users can disconnect their peer to work in an off-line mode and they can add new neighbors in their list to work with. In addition, the UI allows users to see the history of a page, to execute semantic queries and to export the semantic annotations of the wiki pages in an RDF format.

Swooki Manager. The Swooki manager is responsible for the generation and the integration of the editing patches. A *patch* is the set of delete and insert operations on the semantic wiki page (Op = *Insert(PageID, line,* l_P *,* l_N*)* or Op =*Delete(PageID, LineID)*). It implements the Swooki algorithm. Requesting and modifying a page or resolving a semantic query in the RDF repository pass through this manager.

Sesame Engine. The RDF repository used in Swooki is Sesame 2.0 [5]. Sesame is controlled by the Swooki manager for storing and retrieving RDF triples. We used a facility of Sesame to represent RDF triples as multi-set. This component allows also generating dynamic content for wiki pages using queries embedded in the wiki pages. It provides also a feature to execute semantic queries and to export RDF graphs.

Diffusion Manager. The diffusion manager maintains the membership of the unstructured network and implements a reliable broadcast. This component is described in [21, 27].

The integration of the undo mechanism in Swooki requires the addition of the undo component, with slightly extensions of some existing elements. The undo mechanism is designed to allow users to remove or reinsert the effect of some changes in the wiki pages and consequently to update their semantic annotations in the RDF repository. A detailed overview of the proposition is given in the following section.

5 Proposition

We developed the undo component for Swooki to provide users a feature to handle vandalism, to correct errors and to improve easily undesired result of an automatic merge done by Swooki.

5.1 Undo component

In this section, we describe the behavior of the undo component which is responsible of handling undo actions.

When a user wants to undo a modification, i.e. a patch, the document must return to a state in which the modification was never performed according to the undo definition 2. This definition implies two cases:

- the patch is already undone and the document must not be changed,
- the patch must be disabled and its effect must be removed.

Moreover, since the action of undoing a patch is also a modification of the document, users must be able to undo an undo modification, also called redo.

Similarly, according to the undo definition, if a patch is already redone, the action of "redo" has no effect, otherwise, we must re-apply its effect.

As a result, the system must know if a patch is enabled or not. Moreover, the system has to know how many times a patch has been undone or redone as illustrated in figure 4.

Assume that two sites, called Site1 and Site2, have received the same patch $P1$. Concurrently, both sites decide to undo this patch. Consequently, Site1 generates a modification M_1 while Site2 generates M_2. Then, Site2 chooses to redo the patch P1. Finally, each site receives each other modifications. Site1 has received both "undo" modifications and then the "redo". If the system just knows that P1 is undone, the "redo" will reapply P1. Unfortunately, this example violates the definition. Indeed, if the modification M_2 was never produced, the only remaining modification is M_1, then the P1 must remain undone.

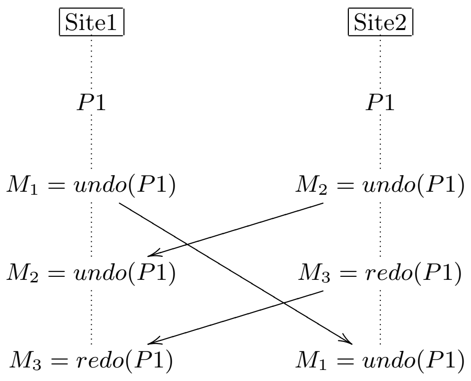


Fig. 4. Concurrent undo and redo messages

For instance, a patch with a *patchDegree* equals to three implies that the patch has been redone three times and that it has an effect in the semantic wiki page model. Actually, patches are never deleted from the *patchGraph*, however their effect is removed from the wiki page model as if they did not exist.

As a result, we propose to extend the *patchGraph* as defined in Swooki by associating a *patchDegree* to each *patch* in that graph. This *patchGraph* becomes a set of <patch, patchDegree> where the *patchDegree* indicates whether the patch has an effect or not in the current state of the wiki page. We arbitrary choose to affect a degree of 1 at the patch reception, to decrease by one the degree at the execution of an undo and increase it by one for a redo. Then, a patch is undone if its degree is strictly inferior to 1.

Consequently, in Figure 4, Site1 will compute a degree of 0 and will not restore P1.

Finally, we translate the description of the undo component above into the following algorithm:

```
IntegrateUndo(patchId):
    patch <− patchGraph.getPatch(message.patchId)
    patchGraph.patch.patchDegree−−
    if patchGraph.patch.patchDegree = 0 then
        disable(patch)
```

```
IntegrateRedo(patchId):
    patch <− patchGraph.getPatch(message.patchId)
    patchGraph.patch.patchDegree++
    if patchGraph.patch.patchDegree = 1 then
        enable(patch)
```

Now, the system can compute whether or not a patch has to be reapplied or undone. Next, we will explain how to remove or reapply a patch.

5.2 Removing and reapplying a patch

Since we keep deleted lines as tombstone in Swooki, we propose to reuse them instead of inserting new lines in case of redo. We call deletion the transformation of a line into a tombstone and, reinsertion the inverse of deletion.

Moreover, using the inverse operations as shown in Figure 2 is not sufficient for removing the effect of a patch. This is mainly due to concurrency: the line "It has a [topSpeed:=49km/h]." is deleted twice (by M_3 and M_4) and "reinserted" once by "Inverse(M_4)". Similarly to the previous example, we propose to count the number of deletion/reinsertion to overcome this issue.

In Swooki, a line in the wiki page model is never deleted, it is only set to invisible in the wiki page view. The visibility of a line is determined through a boolean visibility field. We change the line visibility as defined in Swooki into a visibility degree in order to let the system detect whether a line is visible or not after multiple undo and redo of patches. In our case, a line is visible if it is visibility degree is positive. A delete of a line turns its visibility degree to zero, however an undo or a redo action decreases or increases it respectively by one.

Since we have changed the data model, we have to redefine the operation to modify it. In Swooki, the integration of an operation is processed in two steps: (1) text integration and (2) RDF statements integration. To integrate an insert operation $op = insert(\ PageID, line, l_P, l_N)$, the *line* has to be placed among all the lines between l_P and l_N. Finding the right position where the line should be inserted is done through the Woot algorithm (for more details see [14]).

Once the right position is found, we insert the line in the page with a degree of 1 and insert hypothetical metadata into the RDF store:

```
InsertLine(line) :−
   insert(PageID, line, NextIdentifier)
   IntegrateInsRDF(line)
```

Due to concurrency, a line can have a degree greater than 1. Therefore, the execution of a delete consists in decreasing the degree of that line. If the line becomes invisible, i.e. its degree is now 0, we have to update the RDF store using the method "IntegrateInsRDF".

```
DelLine(LineID) :−
line <− Page[op.LineID]
line.visibilityDegree −−
if line.visibilityDegree = 0 then
  IntegrateDelRDF(line.content)
endif
```

Similarly, the reinsertion of a line increases its degree. If the line is now visible, we update the RDF store.

```
ReinsertLine(LineID) :−
line <− Page[op.LineID]
line.visibilityDegree ++
if line.visibilityDegree = 1 then
```

```
 IntegrateInsRDF(line.content)
endif
```

Finally, we can now remove the effect of a patch:

```
DisablePatch(patch):
   for (op:opsPatch) do
        lineId <− op.getLineID()
        switch(type(op))
          insert : DelLine(lineId)
          delete : ReinsertLine(lineId)
```

or reapply its effect:

```
EnablePatch(patch):
     for (op:opsPatch) do
          lineId <− op.getLineID()
          switch(type(op))
            insert : ReinsertLine(lineId)
            delete : DelLine(lineId)

   }
```

In Figure 2, since the line is deleted twice, its degree is -1. As a consequence, when Site1 undoes M_4 the line degree is increased by one and the line remains invisible.

5.3 Messages

As usual modifications, the undo/redo action must be propagated to all other sites. Therefore, we need to extend the diffusion manager to take account of undo/redo messages.

We define three kind of messages:

Do message: In case of a do message (i.e. containing only insert or delete operations), the patch is added in the *patchGraph* with a *patchDegree* equals to one. All the operations of that patch are integrated depending on their type.

Undo message: In case of an undo message, the patch on which the undo message will be applied is extracted from the *patchGraph*;

Redo message: Similarly for a redo message, the patch is extracted from the *patchGraph*.

When a message is received, we test if it is executable. A *do* message is executable if all its operations satisfy their preconditions as defined in [13], however an *undo* or a *redo* message is executable if the patch on which it is applied exists in the *patchGraph*. When a message is not executable, it is added in a message waiting queue. The message information $mInfo$ of an executable patch is added in the page history. A message is integrated depending on its type.

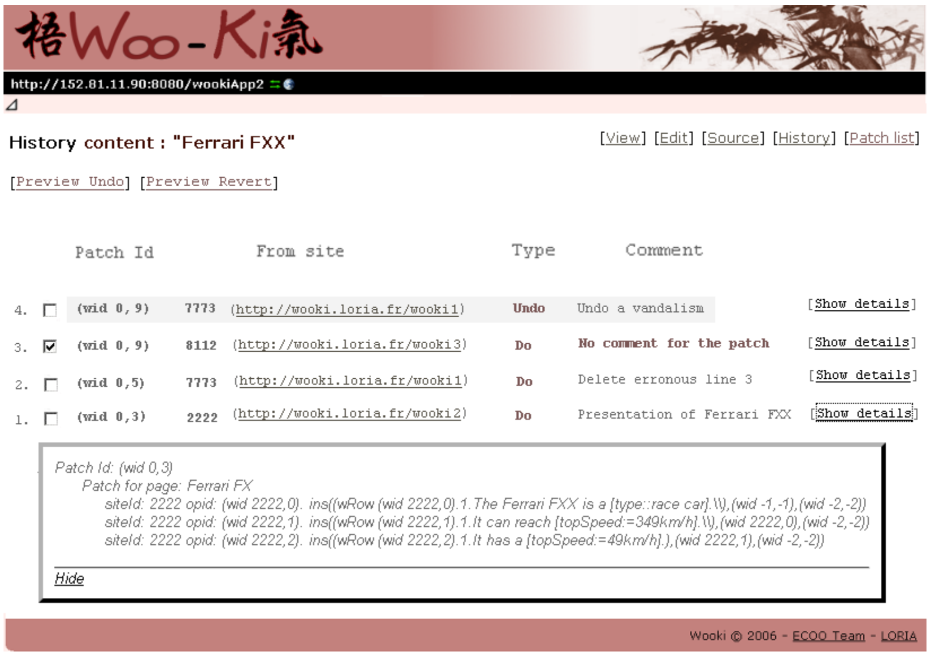
```
Upon Receive(message) :−
 If IsExecuteM(message) {
 History.writeEvent() <− mInfo
 If message.**type** = **do then**
  patch <− message.getPatch()
    patchGraph <− patchGraph $\bigcup$ < patch, 1>
        for (op:opsPatch) do
            if type(op) = insert
              IntegrateIns(op)
            else
              IntegrateDel(op)
else {
patch <− patchGraph.getPatch(message.patchId)
  if message.type = undo then
    DisablePatch(patch)
  else if message.type = redo then
   EnablePatch(patch)

   }
Else
  MsgWaitingQueue <− MsgWaitingQueue $\bigcup$ patch}
```

6 Implementation

The undo or redo of changes can take place either by visiting the history or the patch graph of a page. The figure 5 shows the history where different messages are integrated on the wiki page. A line in the history is equivalent to the message information. It indicates the identifier of the patch, the peer that generated the patch, the type of the message and a user comment when it exists. The undo action of a patch has a grey background. In order to undo a set of patches, a user can click the checkbox that precedes each one of them and then press the undo preview button. The result of this preview is a temporary version of the page which undoes these patches. If he is satisfied, he can apply these changes by saving the undo preview.

Another option provided in the history is to revert the current version of the wiki page. Users can choose to return to a state of a page undoing all the changes integrated after the chosen patch. This is can be achieved by selecting the patch and clicking on the revert preview. Similarly, if he is satisfied the user can save the revert preview and his changes will be applied. The undo of each patch inserts a new line in the history. The history allows also providing more information about each patch by a click on show details link at the end of each line. Each undo action in the history generates a new message of type undo. This message is sent through Swooki diffusion manager to the other peers in order to be integrated. The integration of that message locally or on the other peers is done as defined in section 5.


http://152.81.11.90:8080/wookiApp2

History content : "Ferrari FXX"

[View] [Edit] [Source] [History] [Patch list]

[Preview Undo] [Preview Revert]

		Patch Id	From site		Type	Comment	
4.	☐	(wid 0, 9)	7773	(http://wooki.loria.fr/wooki1)	Undo	Undo a vandalism	[Show details]
3.	✓	(wid 0, 9)	8112	(http://wooki.loria.fr/wooki3)	Do	No comment for the patch	[Show details]
2.	☐	(wid 0,5)	7773	(http://wooki.loria.fr/wooki1)	Do	Delete erronous line 3	[Show details]
1.	☐	(wid 0,3)	2222	(http://wooki.loria.fr/wooki2)	Do	Presentation of Ferrari FXX	[Show details]

```
Patch Id: (wid 0,3)
    Patch for page: Ferrari FX
        siteId: 2222 opid: (wid 2222,0). ins((wRow (wid 2222,0).1.The Ferrari FXX is a [type::race car].\\),(wid -1,-1),(wid -2,-2))
        siteId: 2222 opid: (wid 2222,1). ins((wRow (wid 2222,1).1.It can reach [topSpeed:=349km/h].\\),(wid 2222,0),(wid -2,-2))
        siteId: 2222 opid: (wid 2222,2). ins((wRow (wid 2222,2).1.It has a [topSpeed:=49km/h].),(wid 2222,1),(wid -2,-2))
```

Hide

Wooki © 2006 - ECOO Team - LORIA

Fig. 5. Undo from the history

Another way to undo or redo a patch is through the patch graph (see Figure 6). The patch graph is viewed as an oriented graph of patches. Each patch is a node in the graph and the arrows represent the dependence between these patches. A node relating one or more nodes implies that this patch was generated on a state integrating these patches. Each node is labeled with the patch identifier and the patch degree. A black node is a disabled patch that has no effect in the wiki page. A right click on a node allows to show the information about the patch, to undo or redo a patch or to preview a version of the wiki page. The patch graph is visualized through an applet built using JGraphT java libraries to build the graph model and JGraph to render the graph layout. It is based on a recursive algorithm crossing the patch graph [1]. The patch graph provides information about the patches dependency, hence the concurrency between them.

7 Conclusion

In this paper, we propose an undo mechanism for Swooki. This mechanism allows users to undo any modification at any time, i.e. to remove any modification as if it was never produced. It provides both an undo and a revert features. We developed the undo component, the appropriate algorithms and extended some parts of Swooki to provide an undo mechanism in Swooki. The undo component is

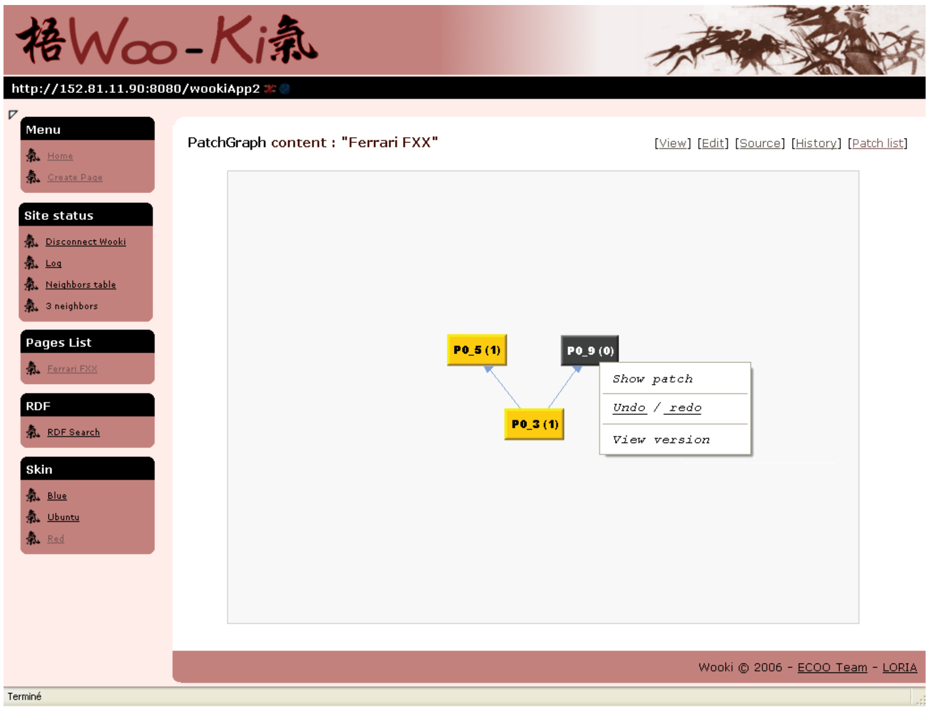


Fig. 6. Undo from the patch graph

responsible for the generation and the integration of undo and redo actions. The propagation of these actions lies on Swooki diffusion manager. Swooki extension ensures the convergence of the wiki pages and the RDF stores on all peers. This convergence is independent from concurrent modifications, the order of integration of the undo or redo actions and the fact that users can edit in an off-line mode i.e. join or leave at anytime.

We identified the problems to resolve in order to achieve such mechanism for P2P semantic wikis. While the revert feature may be sufficient for centralized semantic wikis, this is not the case for P2P semantic wikis aiming at removing any modification at any time.

Our solution is general, it is based: (1) on enabling and disabling patches in the patch graph and (2) on the generation and the integration of undo and redo actions. It can be adopted in any P2P semantic wiki.

As future work, we intend to carry out user studies to evaluate the improvement of the quality of the wiki pages and the knowledge in the RDF stores using our approach and how this mechanism facilitates the task of the users compared to Swooki without the undo mechanism.

References

1. S. Alshattnawi, G. Canals, and P. Molli. Concurrency awareness in a p2p wiki system. In *Proceedings of CTS 2008, The 2008 International Symposium on Collaborative Technologies and Systems, Irvine, California, USA*, May 2008.
2. S. Auer. Powl. In *Proceedings of the 1st Workshop on Scripting for the Semantic Web (SFSW'05), Hersonissos, Greece*, 2005.
3. S. Auer, S. Dietzold, and T. Riechert. Ontowiki - A tool for social, semantic collaboration. In *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 736–749. Springer, 2006.
4. D. Aumueller and S. Auer. Towards a semantic wiki experience - desktop integration and interactivity in wiksar. 2005.
5. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC 2002: First International Semantic Web Conference*, 2002.
6. M. Buffa, F. L. Gandon, G. Ereteo, P. Sander, and C. Faron. Sweetwiki: A semantic wiki. *Journal of Web Semantic*, 6(1):84–97, 2008.
7. B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht. Self-organized Reuse of Software Engineering Knowledge Supported by Semantic Wikis. *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE), November 6th-10th*, 2005.
8. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 399–407. ACM Press, 1989.
9. R. T. Karsten Dello and E. Paslaru. Makna, free university of berlin, 2005.
10. T. Kuhn. Acewiki: Collaborative ontology management in controlled natural language. *In Proceedings of the 3rd Semantic Wiki Workshop, CEUR Workshop Proceedings, 2008*, Jul 2008.
11. F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, Octobre 1989. Elsevier Science Publishers.
12. J. C. Morris. Distriwiki: : a distributed peer-to-peer wiki network. In *Int. Sym. Wikis*, pages 69–74, 2007.
13. G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, Banff, Alberta, Canada, November 2006.
14. G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.
15. A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1(4):295–330, 1994.
16. C. Rahhal, H. Skaf-Molli, and P. Molli. Swooki: A peer-to-peer semantic wiki. In *The 3rd Workshop: 'The Wiki Way of Semantics'-SemWiki2008, co-located with the 5th Annual European Semantic Web Conference (ESWC), Tenerife, Spain*, June 2008.
17. C. Rahhal, H. Skaf-Molli, and P. Molli. Swooki: A peer-to-peer semantic wiki. Research Report 6468, INRIA, 03 2008.

18. M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139, 1999.
19. Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
20. S. Schaffert. Ikewiki: A semantic wiki for collaborative knowledge management. In *WETICE*, pages 388–396. IEEE Computer Society, 2006.
21. H. Skaf-Molli, C. Rahhal, and P. Molli. Peer-to-peer semantic wikis. Research report, INRIA, 2008.
22. A. Souzis. Building a semantic wiki. *IEEE Intelligent Systems*, 20(5):87–91, 2005.
23. C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):309–361, Décembre 2002.
24. C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, New York, États-Unis, Novembre 1998. ACM Press.
25. D. Sun and C. Sun. Operation Context and Context-based Operational Transformation. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 279–288, Banff, Alberta, Canada, Novembre 2006. ACM Press.
26. M. Vôlkel, M. Krtôzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic wikipedia. *Journal of Web Semantics*, 5(4), 2007.
27. S. Weiss, P. Urso, and P. Molli. Wooki: a p2p wiki-based collaborative writing tool. In *Web Information Systems Engineering*, Nancy, France, December 2007. Springer.
28. S. Weiss, P. Urso, and P. Molli. An undo framework for p2p collaborative editing. In *CollaborateCom*, Orlando, USA, November 2008.