Increasing Awareness in Collaborative Authoring through Edit Profiling

S. Papadopoulou, C. Ignat, G. Oster and M. Norrie
Department of Computer Science
ETH Zurich
CH-8092 Switzerland
Email: {papadopoulou, ignat, oster, norrie}@inf.ethz.ch

***Abstract*— Awareness of the activities of other users and also document evolution is an important part of collaborative authoring. We introduce the concept of an editing profile that can be used to maintain and visualise measures of the changes made across a document by different users in both synchronous and asynchronous collaborative editing. It provides a simple means of making users aware of "hot areas" and also who is or has been active in various parts of the document, as well as a quick access point into parts of the document that have been changed. The profile can be adapted to display the most relevant information based on both user preference and situation.**

I. Introduction

Providing awareness has become an important part of improving the usability of synchronous and asynchronous distributed collaborative systems. It is important for people collaboratively authoring a document to be informed about the kinds of changes that have been made on the document between two versions, at which parts of the document they have been made and by which users. Being aware of the changes made to a document helps the user to better understand the evolution of the document, easily cooperate with other users and avoid possible conflicts.

A lot of work has been devoted to the various kinds of information users should be aware of while working in a collaborative environment and how it should be visualised. For example, telepointers and radar views [19], [9] are used to provide awareness information about where users are working on a document. However, little work has been done on computing overall information about changes made by users to a document and how this information should be stored in the system to further be queried and viewed in a flexible way. It should be possible for a collaborative system to provide the users with awareness customised according to their preferences and also their situation. For instance, it should be possible to provide the user with information about insertions of whole words or sentences made by a specific user, at a specific section of the document. Furthermore, it should be possible to present which parts of the document were heavily changed by a specific user, or present an overview of the changes made throughout the whole document.

In current collaborative editing systems, it is difficult to provide flexible awareness information. We propose a novel approach that, given two versions of a text document, either in synchronous or asynchronous collaboration, computes and stores awareness information in a flexible way and presents it to the user at different granularity levels, according to the user's preferences. Our approach is based on the notion of edit profiles which represent a profile of editing operations across a document.

We begin in Section II with a review of existing approaches to awareness in collaborative authoring systems and a motivation of our approach. Section III then presents the requirements imposed by the introduction of edit profiling, while Sections IV and V detail our methods for computing, maintaining and adapting the editing profile according to users' preferences, for different levels of the text document. In Section VI we describe how we extended an asynchronous collaborative text editor to provide awareness information based on our approach. Concluding remarks and a discussion of future work is given in Section VII.

II. Awareness in Collaborative Authoring

When many people work on the same document without the use of collaborative editors, or when a person accesses a document from different computers, it often occurs that the users end up with different versions of the document. Inspecting the differences between two versions of a document is a very difficult process to perform manually.

To solve that problem, many document comparison tools have been developed. Some of them, for example WinMerge [5], diffDoc [2] and diffDog [1], support the comparison and merging of many text-based document types. The procedure followed is that the user chooses the documents to be compared, and the system presents the documents with the differences highlighted.

Some of these systems present information on the user who made each change [4], usually through a colour code used when highlighting the differences, or the type of the changes made (insertion or deletion). Another expedient feature present in WinMerge is a bar serving as an overview of the changes applied to the document, indicating the places where the changes occurred. The users can click on the bar and navigate through the differences of the two documents. At the same time, they are informed about the parts of the documents where the differences occur.

The disadvantage of these document comparison tools is that there is almost no semantic comparison between the

1-4244-0429-0/06/$20.00 ©2006 IEEE

documents. Text is treated as an array of characters, without creating any semantic units in the documents. Even in cases where paragraph, sentence and word separators are specially treated so changes to them will not generate differences between the documents [4], the units are still based on lines and have no semantic value. This is also true in cases where words can be merged to generate a line and lines can be merged to generate a paragraph.

The disadvantage arising from this approach is that no information is, for instance, returned to the user about differences detected at the paragraph level between the documents under comparison. Another drawback of document comparison tools is the fact that most of them are used only to visualise the differences between documents and not to edit the documents. Even in the tools that offer editing features, for example WinMerge [5], the operations performed are not stored by the editor to be further used to compute the difference between the documents. The difference is recomputed, if needed, based only on the initial and final states.

When working on a document using a collaborative editor, there is still the need to present the difference between two states of the document. The advantage of working with a collaborative editor is that the operations that transformed the document from one version to another are recorded by the editor and, for that reason, can easily be stored and used to compute the difference. Many collaborative editors [6], [12], [19], [17], both synchronous and asynchronous, have been implemented based on that concept. The approach of keeping the changes made by users in terms of the operations performed offers better support for merging and conflict resolution compared to state-based approaches [14] since it tracks the activity of a user over a certain period of time. On the other hand, state-based approaches take into account just the final and initial states of the document and lose information about the process of the transformation from one state to the other. We therefore adopted an operation-based approach for computing and storing awareness information during collaborative editing.

Suppose that three users, one professor, Jim, who is a native English speaker and two students, Fred and Mary are using a collaborative asynchronous text editor to write a paper. For reasons of simplicity, we will assume that there is a remote repository in our example where all versions of the document are kept. However, note that the approach presented in this paper can also be used if there is no central repository and users synchronise their versions against other users' versions, or, if they are actually using a synchronous rather than asynchronous collaborative editor. Assume all three users in our example have downloaded the current latest version V_0 from the repository and are working on their local copy. Consider the case that student Fred is mainly responsible for sections 1 and 2, student Mary for sections 3, 4 and the conclusion, and the professor Jim is responsible for the introduction and overall editing of the document.

Assume that while Mary is working on her local copy of the document, Fred makes some changes to his local copy and uploads them to the repository creating a new version V_1. Afterwards, Jim applies some more changes, not conflicting with the ones from Fred, and after updating his version V_0 to include the changes that transformed version V_0 to V_1, he commits his changes to the repository creating version V_2. When Mary updates her version, a log containing all the operations transforming version V_0 first to version V_1 and then to version V_2 is sent by the repository. After these operations have been applied, Mary has an up-to-date local copy of the document including the changes made by the other two users.

This scenario is typical when groups of users are collaboratively authoring a document asynchronously. When a user, such as Mary, updates her copy, she has all the changes integrated in it. However, it is no easy task to locate where and what changes have been applied, especially when the document is large.

We can think of further information of which Mary, in our example, would like to be aware. Since she is not a native English speaker, she would like to be separately informed about spelling mistakes corrected by Jim. This means that she should be informed about changes on different levels of the document (in the case of spelling mistakes the changes are at word level). To see the importance of being aware of changes at different levels, consider further the typical editing scenario where the professor is trying to understand and reformulate a sentence written by one of his students. The number of character edits may become large as he experiments with various reformulations of the sentence, and so, as a measure of the changes made to the student's sections, the number of paragraph or sentences edited might make more sense rather than the number of individual characters edited.

Further, since Mary is mainly responsible for specific sections of the document, she would like to be informed about changes that were applied only to those specific parts of the document, including an indication as to how many insertions or deletions were performed on those sections. Moreover, she might want to check which parts of the document were heavily changed, possibly also which user enforced the changes, and even have an overview of the changes she personally made on the document.

Current collaborative authoring systems concentrate on presenting in detail the changes made to a document. Systems such as Microsoft Word [3], PREP Editor [16] and TeNDaX [11] present all the changes directly superimposed on the text. In some of these systems, the different types of operations are distinguished, as well as the different users who made the changes.

In the TeNDaX approach, the document is represented as a linked list of character objects that are stored in a database. Editing commands for operations of insertions and deletions of characters are mapped to database transactions. TeNDaX does not provide any awareness information associated with the structure of the document, such as an overview of the changes made on a certain section or paragraph of the document.

The PREP editor supports visualisation of changes at different document levels. However, PREP editor, similarly to

Microsoft Word, does not provide a global overview of the changes made by users throughout the document. Users have to scroll through the document to visualise the parts that have been modified. This requires explicit actions on behalf of the user and will be tiresome in the case of very large documents. Therefore, there is a need for an auxiliary, less detailed, representation, alongside the main text, that shows a measure of the changes made across the document, thereby providing a quick and easy way to make users aware of "hot areas" with lots of changes, the activities of different users and, if shown over a longer history, also the evolution of the document.

A collaborative editing system should monitor user activity and record all changes made to various parts of the document. A summary of all edits would then form a global editing profile. The scale for the representation of changes should be flexible, allowing it to be adapted according to specified document levels of sections, paragraphs, sentences and words. Users should also be allowed to use the global editing profile to select parts of the document and visualise the changes made there.

Such an editing profile could also provide better awareness for users collaboratively editing a document in real-time. Consider, for instance, an example of two users using a real-time collaborative editor. The editing profile provides each user with an overview of the activity of the other users in terms of both the quantity and location of changes made. Returning to our previous example, Jim may be working sequentially through the document correcting language, while Mary works on the implementation section. In real-time, Mary will be aware of Jim's progress through the document and see when he is approaching the section on which she is working.

The idea of profiles attached to a document can also be found in information retrieval. Generally, an overview of the retrieved documents for a given query is provided as a ranked list of documents. In [7], [8], a new overview for within-document retrieval was introduced in terms of a relevance profile that enables users to identify relevant parts of a document with respect to a query and then select those parts for browsing. A relevance profile is presented in the form of an interactive bar graph. Each bar refers to a part of the document and the height of a bar represents the computed *retrieval status value* [7] of the corresponding part of the document. However, the unit for visualising changes is the so called *tile* representing a part of document that consists of a fixed number of words. Therefore, the granularity unit cannot be varied dynamically to specify the relevance measures for different semantic levels within the document, such as paragraph, sentence or word.

The idea of presenting an overview of a project's evolution is not new in CSCW. By analysing differences between multiple revisions of one document, history flow visualisations [18] provide an overview of a document's evolution. They provide information about how a group has contributed to a document or how a modification has influenced the current version of a document. Unfortunately, this work relies on a state-based difference approach and the document's evolution is computed only on the document level.

Molli et al. [15] proposed a metric to measure divergence between copies of the same document. By informing users how their copies are diverging from each other, and presenting a measure of the conflicts that the changes will cause when published, it is expected to generate auto-coordination in a group working collaboratively. Although this approach seems to be promising, the unit for computing the divergence is the document, rather than semantic units of the document. Therefore, it is not possible to provide users with a detailed view of the modifications performed on document parts.

The idea of *computational wear* [10] is the closest work to our proposal. Actions of reading and authoring lines of a text document are graphically depicted. This is achieved by counting how many times a line is read or updated, and presenting this information to the user as bar charts drawn in the editor scrollbars. The lack of a structured document model restricts considerably the accuracy of the information provided. No flexible way is provided to filter the above information according to user preferences and present it on different granularity levels.

We propose a flexible way of computing awareness information on different semantic levels of a text document. Operations applied to a document, either locally or remotely, are scanned through and used to evaluate some new metrics that we introduce in this paper. These metrics will represent the amount of changes on specific parts and levels of the document. This information can further be easily filtered to:

- provide an overview of the changes applied throughout the document or to a part of it,
- inform the user about different types of changes applied to the document together with the part of the document where they were applied,
- give an overview of the changes made on the document by specific users,
- indicate which parts of the document were heavily changed.

The maintenance of editing profiles requires some changes to the representation of documents and their operation history in a collaborative editing system and we present these in the next section before going on to detail the metrics that we use.

III. System Requirements for Edit Profiling

Providing information at different granularity levels about the changes applied to a document requires a flexible document model. We use the hierarchical document model proposed in [12] instead of a linear document model, thereby allowing documents to be accessed on different levels. Consider, for instance, the case of a book containing chapters composed of sections. Each section is composed of paragraphs, each paragraph of sentences, each sentence of words and each word of characters. In this case the granularity levels associated to the hierarchical model would be book, chapter, section, paragraph, sentence, word and character. The same hierarchical model can be used also in the case of XML documents with n granularity levels as proposed in [13].

Based on this, we now present the notions of *node* and *operation* used in our collaborative system.

Definition 1 (node) *A node N is a structure of the form*

$$N =< level, children, length, history, content >$$

- *$level$ is a granularity level, $level \in \{0, 1, ..., n\}$, corresponding to the element type represented by the node N. Granularity level n corresponds to character nodes.*
- *$children$ is an ordered list of nodes $\{N_1, ..., N_m\}$, such that $\forall i \in \{1, ..., m\}$ $level(N_i) = level(N) + 1$*
- *$length$ is the length of the node,*

$$length = \begin{cases} 1 & \text{if } level = n \\ \sum_{N_i \in children} length(N_i) & \text{otherwise} \end{cases}$$

- *$history$ is an ordered list of already executed operations on nodes in $children$*
- *$content$ is the content of the node, defined only for leaf nodes*

$$content = \begin{cases} \text{undefined} & \text{if } level < n \\ \text{a character} & \text{if } level = n \end{cases}$$

For generality, we also define a node at the character level, setting the *history* and *children* elements as empty sets.

Definition 2 (operation) *An operation op is a structure of the form*

$$op =< level, type, position, content, userID >$$

- *$level$ is the granularity level of the operation, $level \in \{1, 2, ..., n\}$,*
- *$type$ is the type of the operation, $type \in \{insertion, deletion\}$,*
- *$position$ is a vector of node indexes, $position[i]$ is the node index for the ith granularity level, $i \in \{1, \ldots, level\}$,*
- *$content$ is the node inserted or deleted by the operation,*
- *$userID$ is the identifier of the user who executed the operation.*

Operations of $level$ n reflect insertions and deletions of whole characters. The insertion or deletion of the whole document is not permitted. Note that the level of an operation is the level of the node N inserted or deleted and that the operation is kept in the history associated with the parent of node N. The vector $position$ specifies the indexes that compose the path in the tree where the operation is applied. For example, if we have an insertion operation of word level, we have to specify the sentence, the paragraph, the section etc. in which the word is located, as well as the position of the word inside the sentence. The $content$ of an insertion operation specifies the node to be inserted at the position given by the $position$ vector. In the case of deletion, $content$ can be used to support the undoing of operations.

For the sake of simplicity, in the rest of the paper, we assume that a document is composed of paragraphs, sentences, words and characters, thus containing only 5 different granularity levels. In Figure 1 an example of a document with the 5 levels of granularity - document, paragraph, sentence, word and character - is illustrated. We assign numeric values to each granularity level, as follows: for the document level, value 0; for the paragraph level, value 1; for the sentence level, value 2; for the word level, value 3 and, for the character level, value 4. As shown in Figure 1, a history is assigned to each node, containing the operations performed on the children of that node. For example, the log of operations associated with a paragraph will include insertions and deletions of sentences in that paragraph.

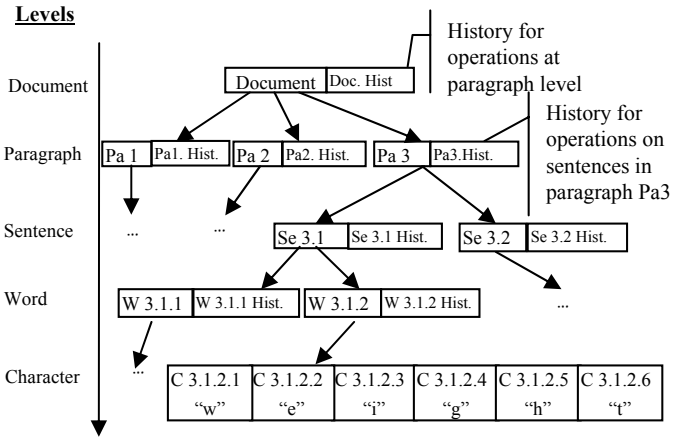

Fig. 1. The hierarchical document model

Moreover, in what follows, we denote operations by specifying only their level, type, position and the text conversion of content, ignoring other attributes. For example, $insertWord([2, 3, 2], \text{“}awareness\text{”})$ denotes an operation of type $insertion$ that is at word level and has to be applied to paragraph 2, sentence 3, at word position 2, and has as content a word node represented by the string “$awareness$”. Note that in our simplified notation the type and the level of the operation are expressed by the name of the operation.

An advantage of using a hierarchical structure for the document model is that we can compute awareness information for different semantic levels of a document. A further advantage is that the operations performed on a document are not kept in a single history buffer, but instead are distributed throughout the tree, associated with the nodes they are targeting. Compared to other approaches that use a single history buffer, the distribution of the history offers higher efficiency and the possibility of querying information at different levels of granularity [12].

IV. Computation and Maintenance of Profiles

In this section we present details of the information computed for the editing profile and also how we compute it. Throughout the section, we will use the example presented in section II of Jim, Fred and Mary writing a paper together with the help of an asynchronous collaborative text editor. At any time, the three users can download the latest version of the document from the repository and work on it. They may

also update their local (older) version, at any time, to arrive at the latest version in the repository, by including the changes that were applied by others. Note that the same ideas and computations can be applied in the case of a synchronous collaborative editor.

Every time a user performs an update of their local copy, a set of operations are sent from the repository to be applied to their local document version. Having chosen to work with a structured document model and a distributed history, the set of operations to be applied to the document are the sum of the operations that need to be applied to each node separately. In the same manner, local changes made by the user, are applied to the document and recorded in the history of the node where the operation was applied.

The sum of operations that need to be applied to each node, either the local ones, or the remote ones transformed against the local ones to include their effect [6], represent the changes made to the node between the two states of the document. We need to define a metric that measures the effect of each operation on the node. For that reason, we define the metric $opWeight$ as follows.

Definition 3 (opWeight) *$opWeight$ is defined as the length of the operation's content, i.e. the number of characters inserted or deleted by the operation.*

$$opWeight(op) = length(content(op))$$

We decided to use the length of an operation's content as the metric that shows how much the operation affects the node. We believe that this item of information is the one that the user would expect to be used, as it is the most intuitive one. It could be argued that operations vary in importance depending on the user performing them. While we agree that this may be the case in some situations, we believe it is preferable to have a metric that uniformly evaluates changes made to a document. Once the changes are uniformly evaluated, we provide the user with the means to query that information according to his preferences and the context, for example, to see only the changes made by a specific user.

However, as we report later in the section on future work, we plan to investigate the possibilities of extending the metrics to compute the changes made in a document based on insertions and deletions of higher level units of the document model. For example, we could also compute the changes that a user makes, such as the number of sentences or paragraphs inserted, and not just the changes in terms of the number of characters inserted.

After introducing the $opWeight$ metric, we define the $nodeWeight$ metric to provide information about the changes made on a document node. The $nodeWeight$ has to include the weights of all the operations applied to the node. However, changes made to the node's children represent changes made to the node as well. Due to the fact that we use a distributed history of operations, operations concerning changes to a node's children are kept on the children's logs and not on the node's log. For that reason, we add a second factor to the $nodeWeight$ metric, defined below, representing the changes made to each of the node's children.

Definition 4 (nodeWeight) *We define $nodeWeight$ as the sum of two components: the sum of the $opWeight$ of all the operations applied to the node and the sum of the $nodeWeight$ of all the node's children.*

$$nodeWeight(N) = \sum_{op_i \in history(N)} opWeight(op_i) + \sum_{N_j \in children(N)} nodeWeight(N_j)$$

Note that the $nodeWeight$ of a node at the character level is equal to zero, since its history and children are empty sets.

For every node, we additionally define an $insNodeWeight$ and a $delNodeWeight$ as the $nodeWeight$ metric computed for only the insert and delete operations, respectively.

Definition 5 (insNodeWeight) *The $insNodeWeight$ is defined as the sum of two components: the sum of the $opWeight$ of all the insertion operations applied to the node and the sum of the $insNodeWeight$ of all the node's children. The $delNodeWeight$ is defined in a similar way.*

$$insNodeWeight(N) = \sum_{\substack{op_i \in history(N),\\ type(op_i)=insert}} opWeight(op_i) + \sum_{N_j \in children(N)} insNodeWeight(N_j)$$

For each node we create a table $weights$ containing information about the identifier of a user, $userID$, an $insNodeWeight$, and a $delNodeWeight$. Every row in the table holds information about the number of characters inserted at and deleted from that node and its descendants by the user with identifier $userID$. The information kept in the table is updated each time a new operation is applied to that node or its descendants. For instance, Table I holds the following information for our editing example. Jim has inserted 23 characters and deleted 15 at the node under investigation, whereas Mary performed only insertions (18 characters) and Fred only deletions (4 characters).

TABLE I
Weights TABLE FOR A SPECIFIC NODE

userID	*insNodeWeight*	*delNodeWeight*
Jim	23	15
Mary	18	0
Fred	0	4

In what follows, we will demonstrate with an example the computation of the metrics defined above. Consider the case where Fred and Mary collaboratively edit the latest version V_n of a document. Suppose that the first sentence of the third paragraph of this document is "$We\ assign\ a\ weigh\ every\ document\ node$." and that Mary performs the following operations. She first notices

the spelling mistake in the word "$weigh$" and corrects it by adding the character "t" at position 6. Then she inserts the word "to" to be the 5th word of the sentence. The operations generated are: $op_1 = InsertCharacter([3,1,4,6],'t')$ and $op_2 = InsertWord([3,1,5],$"$to$"$)$ meaning insert the character "t" as the 6th character of the 4th word of the 1st sentence of the 3rd paragraph and insert the word "to" as the 5th word of the 1st sentence of the 3rd paragraph. The structure of the document after the execution of these operations is shown in Figure 2. Due to the distributed history, operation op_1 is kept at the word-level node's history while op_2 is kept at the sentence-level node's history. Mary commits the changes to the repository creating a new version V_{n+1} to be stored as the latest in the repository. Fred then updates his local version. During the update procedure, a log containing the operations that transformed version V_n of the document to version V_{n+1} is sent to him, i.e. the two operations generated by Mary. Afterwards, one by one, the operations are executed locally. Suppose that the current value of each node's $insNodeWeight$ is equal to zero. Operation op_1 is applied and the value of the $insNodeWeight$ for the word $W3.1.4$ is updated to have value 1, equal to the number of characters inserted in the word. The $insNodeWeight$ of all the ancestor nodes is also updated to have the value 1. In the same way, when operation op_2 is applied, the $insNodeWeight$ of the sentence-level node $Se3.1$ is updated to 3, reflecting the sum of all the characters that have been inserted into the sentence. The final step is to update the $insNodeWeight$ of the ancestor nodes of sentence $Se3.1$. The final values of all the document nodes' $insNodeWeight$ are shown in Figure 2.

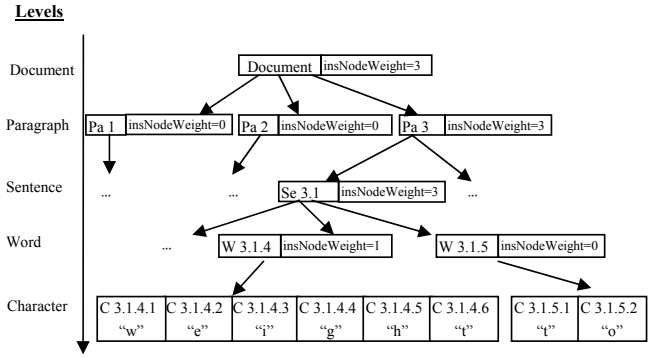

Fig. 2. Computation of *insNodeWeight* throughout different document levels

The above procedure is applied throughout the entire document. Once an operation is applied at a node and the node's weights are up-to-date, we traverse the tree document in a bottom up manner, updating the weights of all the ancestor nodes to include the effect of the new operation. This is achieved by adding to the values of a node's weights the corresponding weights of its child nodes.

Note that in the process of writing or reviewing a document, there is a special case that needs to be taken into account. As mentioned earlier, it is often the case that a user cannot decide immediately what to write or how to express an idea. As a result, the user repeatedly inserts and deletes characters until they are satisfied with what is written. This retrogression causes the generation of many insertion and deletion operations in the specific part of the document, or equally at the nodes corresponding to that part of the document, although, in effect, they have performed a single operation such as the insertion of a new sentence. Applying all the operations performed would result in large values for the insertion and deletion weights of the corresponding nodes, although this does not really reflect the difference between the two states of the document, especially since many of these operations nullify each other when applied to the document.

In the case of local operations performed using an asynchronous editor or local and remote operations using a synchronous editor, the operations that nullify each other are applied in real time, and for that reason we take them all into account when computing the weights. We believe that an overload of operations on a specific part of the document is information that should be provided to the users in such situations. In the first case (asynchronous collaboration), the users will be informed about the amount of effort they or other users invested into that part of the document, and in the second case (synchronous collaboration), the users will be notified that a lot of changes occurred at that part and may, for instance, decide not to work on the same part of the document in order to avoid conflicts.

However, when we compute the effect on the nodes' weights from remote operations in an asynchronous editor, we believe that the operations that nullify each other should be handled in a different way. It can be argued that all operations are needed to provide maximum awareness to the users. Even operations that nullify each other should be taken into account when computing the weights for awareness. In such a case, the awareness information computed would reflect the amount of time and effort that a user spent on a document, even if they later decide to discard some of their own changes.

Nevertheless, it can be also argued that, when updating a document in asynchronous collaboration, the users are working in privacy and for that reason they may want to publish only the part of the changes that they have not further cancelled. To respect the users' privacy, it is reasonable to provide them with the right to decide whether all the changes that they made in the document will be published, or only the ones that were not discarded afterwards.

For that reason, we decided to provide two options when publishing local changes to other users. If the users want to publish only the changes that were not later discarded, they need to first compress the log with their changes and then publish. Log compression is a procedure that discards from the log, all the operations that nullify each other. Otherwise, they can publish the uncompressed log, providing other users with information about changes that they applied but later discarded. Of course, to make it more convenient for the users, these two possibilities can be specified as general default preferences.

V. Adapting the Profile to User Preferences

Computing the weights for each node and storing them has the advantage that we can further filter them according to users' needs and present awareness information in a customised editing profile at different granularity levels of the document. Keeping separately the $insNodeWeight$ from the $delNodeWeight$ has the advantage that information about insertions or deletions on a specific node can be presented separately. The users can decide whether they want to have an overview of insertions in the document, or deletions, or both of them.

Using a structured document model enabled us to compute the different weights for each node at each level of the document. This provides the system with flexibility to present changes on different levels of the document. Suppose that a user chooses to be informed on the paragraph level about changes, i.e. how much each paragraph was modified. In such a case, the system will return the corresponding weights of the nodes on the paragraph level.

It is also possible for a user to require information about the document's changes at a more detailed level as well. By choosing a specific node, the tree document (with the weighted nodes) is appropriately filtered and the system provides all of the above information for the corresponding sub-tree of the document.

Summarising the above, we provide the user with the possibility to visualise changes:

- on specific types of operations,
- applied at a specific document level,
- applied by specific users,
- applied to a specific part (node) of the document.

We define two functions that filter the weights of all the nodes in the tree and return only the required weights for the corresponding node or level of the tree document for specific types of operations and users.

To illustrate the need to define awareness for a specific level or node of a document, let us consider the following example. Mary, after updating her local version, wants to see how much each paragraph in the document changed. Therefore, she only needs to specify the level in which she is interested, namely paragraph. Suppose now that she is interested in changes made to a particular document node, in terms of the changes made on finer granularity units belonging to that node. For instance, she could be interested in changes made on sentences belonging to a specific paragraph. Filtering awareness information on the sentence level would provide the user with the weights of all sentences in that document. To avoid this, we also allow the user to filter awareness information by specifying only a certain document node. Based on the user's choice, either the $awarenessOnLevel$ function or the $awarenessOnNode$ function is called.

Definition 6 (awarenessOnLevel) *We define the function* $awarenessOnLevel(opType, level, userList)$*, where*

- $opType \in \{insertion, deletion, both\}$ *is the type of the operations that the user wants to visualise,*
- $level \in \{1, 2, ...n-1\}$ *is the level in the document tree where the changes should be applied to be visualised,*
- $usersList$ *is a list of users whose changes should be visualised,*

that filters the weights of the nodes on document level $level$ *and returns the weights corresponding to users* $usersList$ *and operations with type* $opType$*.*

Definition 7 (awarenessOnNode) *We define the function* $awarenessOnNode(opType, node, userList)$*, where*

- $opType$ *and* $usersList$ *are defined as in the* $awarenessOnLevel$ *function*
- $node$ *is the document node whose changes should be visualised,*

that filters the weights of the node's $node$ *children and returns the weights corresponding to users* $usersList$ *and operations with type* $opType$*.*

The functions defined above can be used to extract information about users or even roles in collaborative situations. The information extracted directly are the changes that a specific user has made. In our example, Mary could decide to separately see the changes made by Jim, since these changes could be of greater importance.

However, the information extracted alongside is equally important. Consider a collaborative situation, where each user has to write one section of the document and review another. It could be interesting to recognise users or user roles based on the parts of the document that they are editing, or based on the patterns that the users use when authoring or reviewing a document.

This information is more direct and can be extracted more easily in the case that the computation of weights is applied in a synchronous editor where the weights are continuously updated and presented to the user in real-time. Seeing, for example, that the weights corresponding to the second section of the document are increasing, the user is aware that another user is working on that part of the document. If the weights of all the sections are slightly increasing one after the other, it may mean that a reviewer is going through the whole document making small changes everywhere. We realise that the potential use of the editing profile that we compute is greater than the ones presented in this paper, especially concerning synchronous collaboration, and are investigating other ways of exploiting this information.

The information returned by the functions presented above can be visualised in many different ways. One advantage of our generic model is that it enables us to easily experiment with different visualisations in terms of both information content and presentation. In the next section, we present a prototype of the asynchronous editor we have built, enhanced with a visualization of editing profiles.

VI. Implementation

The asynchronous text editor that we have developed is based on the hierarchical document model presented in section III. We have extended the editor to include:

- the computation of the $insNodeWeight$ and the $delNodeWeight$,
- the collection of the input parameters for functions $awarenessOnLevel$ and $awarenessOnNode$ as user defined parameters,
- the visualization, as an interactive histogram, of the weights returned by the functions.

When the user updates their local version, they receive the changes made from other users. While applying the changes to their local copy, the $insNodeWeight$ and $delNodeWeight$ are computed for every node, as presented in section 4. Further, the information collected can be filtered by the user according to their preferences via the editor's graphical user interface (GUI).

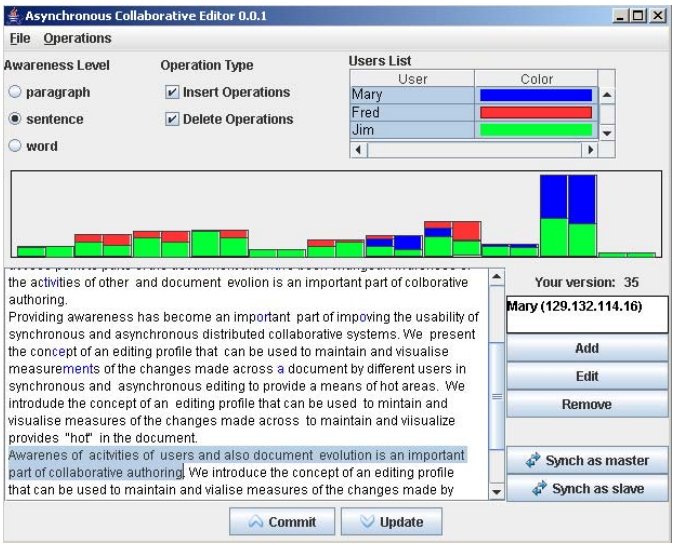

Fig. 3. Awareness enhanced GUI

We have extended the GUI to include the features shown in Figure 3. Note that the editor was created based on the simplified document model consisting of the document, paragraph, sentence, word and character levels. Therefore, we provide the user with the possibility of being aware of changes on paragraph, sentence and word level, i.e. visualise the weights of document-level, sentence-level and word-level nodes. Further, they have the possibility of choosing the type of changes that they want to visualise. They can choose insertions, deletions or both. Additionally, we provide a table with an entry for every user who has made changes to the document. A colour can be assigned to a user and this is used in the histogram, as described below, to distinguish the changes according to the user who made them. A user can select a row in the table, to specify an individual user whose changes they want to visualise. Multiple selections of rows, i.e. users, is also possible.

After specifying the document level, the operations type and the users, the weights of the tree document are filtered as described in section 5, and the results presented in the histogram. A bar is drawn for every node, showing the number of changes that were made to this node and its children. In the case that only insertions or only deletions are selected, each bar shows the number of insertions or deletions, respectively, made to the corresponding node. Each bar is coloured according to the user who made the changes. If more than one user is selected, each bar is horizontally split into subbars. The height of a subbar is relative to the proportion of the user's changes in the total number of changes made to the corresponding node. If both insertions and deletions are selected, each bar is vertically separated into two subbars to present the insertions and deletions separately. The left subbar shows the number of insertions and the right subbar the number of deletions. If additionally, more users are selected, the subbars are further horizontally split and coloured as presented above.

In the rest of this section we will demonstrate, with the help of an example, how the users can use the awareness mechanism of our asynchronous editor. Suppose that Fred, Jim and Mary are working collaboratively on a document with 3 paragraphs. All users have a local copy of the same document's version and start working on it. Suppose as well that, after all three users have stopped working on the document and committed their changes, the editing profile in Mary's GUI is the one shown in Figure 3.

Mary has chosen to see changes made at the sentence level by all three users. The information displayed in the histogram can be interpreted as follows. Fred has worked only on the second half of the document and concentrated mainly on the last paragraph, where he made most of the changes. Jim has worked on the whole document making small changes in each part of it. It could be inferred that he was checking for spelling mistakes. Mary can also see an overview of her own changes on the document. She can also detect parts of the document that were heavily changed and also see the proportion of a user's changes in the total changes in a node. For instance, she can see that the part of the document that changed most is the last paragraph and that Fred and Jim made almost the same number of changes to it.

In order to harmonically associate the information displayed in the histogram with the parts of the text document where the changes were made, we made the histogram interactive. The user can click on a bar in the histogram to obtain detailed information about the node corresponding to the bar. With a single left-click, the text corresponding to the node is highlighted. With a double left-click, the histogram displays the weights of the node's children. In this way, the user can be informed as to how the total number of changes made to that node is distributed to the node's children. In our example, Mary has clicked on the bar with the most changes and the corresponding sentence is highlighted in the text.

When the user is redirected from the histogram to the text, they should be able to easily identify the changes made in a specific part of the document. For that reason, we additionally mark the changes made on the document by highlighting the corresponding characters in the text. All the changes made to the document are highlighted, even if they are from different users. We plan to further extend that feature to

differentiate changes made by different users by highlighting them according to the colour code used in the histogram and also extend our prototype by providing a more user-friendly interface. Summarising the above, it is clear that, by using the editing profile, a user can easily and quickly have an overview of the changes made to the document, be informed about parts of the document where each user was active, be aware of "hot" areas in the document and navigate quickly through the changes made to the document without the need of a scrollbar.

VII. Conclusion

We have presented a novel approach to increase awareness in synchronous and asynchronous collaborative authoring tools, based on the notion of editing profiles. These profiles provide a quick and easy way to inform users about parts of the document with many changes. It can also be seen as an overview of the activity of other users in terms of the quantity and location of changes within a text document.

We introduced metrics for computing awareness information on the different parts and levels of a hierarchically-modelled text document as well as ways of filtering this information according to user preferences. Changes made on the document can be filtered according to their type, the user who made them and the level or part of the document where they were applied. We have extended the prototype of an asynchronous editor to include the computation and visualisation of the editing profile.

We are currently investigating possible extensions of the metrics. We would like to compute the changes made to a document based not only on insertions and deletions of characters, but also of units at a higher level in a hierarchical document structure. Furthermore we plan to extend a synchronous collaborative editor to provide editing profiles and then carry out user studies with both the asynchronous and synchronous collaborative editors. Moreover, we plan to extend and apply the metrics to other types of hierarchical documents, such as XML or graphical documents, where more complex changes than insertions and deletions of nodes occur.

We additionally plan to study the benefits of using profiles in our asynchronous editor to provide information in real-time about changes made to the document by other users. In this way, the user will continue to work on their local copy in isolation, and at the same time be informed via a continuously updated profile about activities of other users in various parts of the document.

References

[1] Altova DiffDog, XML-aware differencing and merge tool. http://www.altova.com/diffdog/.

[2] Diff Doc, the comprehensive document comparison tool. http://www.softinterface.com/MD/Document-Comparison-Software.htm.

[3] Microsoft Word. http://office.microsoft.com/word/.

[4] SubVersion, an open source version control system. http://subversion.tigris.org/.

[5] WinMerge, an open source visual text file differencing and merging tool. http://winmerge.sourceforge.net/.

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

[7] D. J. Harper, S. Coulthard, and Y. Sun. A Language Modelling Approach to Relevance Profiling for Document Browsing. In *Proceedings of the ACM/IEEE-CS joint conference on Digital libraries - JCDL 2002*, pages 76–83, Portland, Oregon, USA, 2002. ACM Press.

[8] D. J. Harper, I. Koychev, and Y. Sun. Query-Based Document Skimming: A User-Centred Evaluation of Relevance Profiling. In *Proceedings of the european conference on Information Retrieval Research - ECIR 2003*, volume 2633 of *Lecture Notes in Computer Science*, pages 377–392, Pisa, Italy, 2003. Springer Berlin / Heidelberg.

[9] S. Hayne, M. Pendergast, and S. Greenberg. Gesturing Through Cursors: Implementing Multiple Pointers in Group Support Systems. In *Proceeding of the Hawaii International Conference on System Sciences - HICSS'93*, pages 4–12, Maui, Hawaii, Jan. 1993. IEEE Press.

[10] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit Wear and Read Wear. In *Proceedings of the ACM SIGCHI conference on Human Factors in Computing Systems - CHI'92*, pages 3–9, Monterey, California, USA, May 1992. ACM Press.

[11] T. B. Hodel-Widmer and K. R. Dittrich. Concept and Prototype of a Collaborative Business Process Environment for Document Processing. *Data & Knowledge Engineering*, 52:61–120, 2005.

[12] C. L. Ignat and M. C. Norrie. Customizable Collaborative Editor Relying on treeOPT Algorithm. In *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2003*, pages 315–334, Helsinki, Finland, Sept. 2003. Kluwer Academic Publishers.

[13] C.-L. Ignat and M. C. Norrie. Flexible collaboration over xml documents. In *Proceedings of the Third International Conference on Cooperative Design, Visualization and Engineering (CDVE'06)*, pages 267–274. Springer Berlin / Heidelberg, 2006.

[14] E. Lippe and N. van Oosterom. Operation-Based Merging. In *Proceedings of the ACM SIGSOFT Symposium on Software Development Environments - SDE 5*, pages 78–87, Tyson's Corner, Virginia, USA, Dec. 1992. ACM Press.

[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, 2002. Society for Design and Process Science.

[16] C. M. Neuwirth, R. Chandhok, D. S. Kaufer, P. Erion, J. Morris, and D. Miller. Flexible Diff-ing in a Collaborative Writing System. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW'92*, pages 147–154, Toronto, Ontario, Canada, Nov. 1992. ACM Press.

[17] H. Shen and C. Sun. Flexible Merging for Asynchronous Collaborative Systems. In *Proceeding of the International Conference on Cooperative Information Systems - CoopIS'02*, volume 2519 of *Lecture Notes in Computer Science*, pages 304–321, Irvine, California, USA, 2002. Springer-Verlag.

[18] F. B. Viégas, M. Wattenberg, and K. Dave. Studying Cooperation and Conflict Between Authors with History Flow Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI 2004*, pages 575–582, Vienna, Austria, 2004. ACM Press.

[19] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging Single-User Applications for Multi-User Collaboration: The CoWord Approach. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW 2004*, pages 162–171, Chicago, Illinois, USA, Nov. 2004. ACM Press.