VTML for fine-grained change tracking in editing structured documents

Lars Bendix[1] and Fabio Vitali[2]

[1] Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark
bendix@cs.auc.dk
[2] Computer Science Department, University of Bologna, Mura Anteo Zamboni 7, I-40121 Bologna, Italy
fabio@cs.unibo.it

Abstract. The task of creating documents collaboratively is complex and it requires sophisticated tools. Structured documents provide a semi-organised writing environment where collaboration may assume more controlled forms than with other document types. CoEd is a writing environment that provides integrated structure support, content overview and version management for complex and hierarchical documents (e.g. technical documentation). The present implementation has, however, limitations in the efficient management of disk usage and in providing more sophisticated collaboration functionality. This led us to consider the VTML change tracking language as a backend for improving the performance and feature set of CoEd. This paper explores the advantages of using a sophisticated change-tracking language in a versioning system for collaborative writing.

1 Introduction

Collaborative writing of documents requires great care and discipline from the participants to avoid coordination problems. On the other hand extreme flexibility must still be possible in what can be a highly creative and unpredictable process. This poses great demands on tools that have to support such work. Organising the document in a hierarchical way provides enough structure to control how the collaboration develops. Under such conditions environments can make assumptions that make the task easier.

The CoEd system [BLNP98] was born to provide support for collaborative writing to teams of students at the University of Aalborg needing to prepare Latex reports for software projects connected with their courses. The available tools were felt lacking in facilities for global overview, co-ordination, version control and communication among writers.

The first prototypes [BNLP97] built at the University of Aalborg provided overview of the structure of the texts and version management of the basic text units of the students' reports. Thus, CoEd was able to solve many problems connected to the mentioned tasks. Students could carry out satisfactorily the processes connected to

writing their reports and the availability of a sophisticated tool such as CoEd reduced the efforts for creating and correcting them.

On the other hand, CoEd implements an unsatisfactory management of persistent data, by storing whole versions and ignoring the inherent structure of the documents handled. Furthermore, many useful features are not implementable given the current underlying data model, such as support for managing and visualising differences in the structure, querying of attributes or comparison of parallel versions. Evolving the CoEd system to handle this kind of information and to provide the functionality requires deciding on a mechanism for improved internal data management.

VTML (Versioned Text Markup Language, [VD95]) is a markup language for describing changes occurred to arbitrary sequential data types. It allows to specify arbitrary attributes to each change, such as authors and dates, and to build arbitrarily complex version graphs detailing the development of a document. A VTML-based system rely on the VTML format to provide support for efficient data management, version branching, lock-free concurrent access to shared documents [SVWD98], version identification, easy comparison of versions, and reliable addressing space for the document's content.

VTML seems adequate for providing intelligent data management to CoEd, and to provide support for much of the sophisticated functionality mentioned. Other engines were judged inadequate because they are not capable of dealing with structured texts as easily as VTML can. Additionally, the generality of VTML with regard to versioning policies allows an easy adaptation to CoEd's specific policies and collaboration styles among the team members contributing to a document. Finally, additional information, in the form of attributes, is handled in a straightforward way by VTML.

We will create a new prototype for handling change tracking in collaborative writing efforts. It will rely on the old CoEd to provide the user interface and the versioning model, behaving as the front-end of the system, and use VTML to provide the versioning engine and work as the back-end. The new prototype is supposed to give better change tracking than previous systems and better change tracking helps improve the co-ordination and communication in collaborative writing efforts.

In this paper, we describe the goals of our current research project aimed at:

- specifying the requirements for a collaborative writing environment
- specifying the interface between CoEd and VTML
- designing the new integrated prototype
- implementing the services that were not implemented with the old back-end
- testing the flexibility and generality of VTML through the implementation of specific versioning policies

The rest of the paper is structured as follows: In section 2, we describe the problem area and the existing CoEd system and the results that were obtained by using it. In section 3, the VTML language is described and a simple session is analysed to provide insight into the actual working of the language. Section 4 describes the analysis of the problems in implementing CoEd policies using VTML mechanisms and sketches the design of the integrated system. In section 5 we draw some conclusions, draft our plans for future work in the project and state some preliminary results.

2 Collaborative Writing

CoEd is a collaborative tool aimed at supporting teams in writing shared structured documents. For years the students at the Department of Computer Science at Aalborg University have encountered numerous problems when they had to work together to write reports. Each semester these students spend the major part of their time developing a system, enabling them to put the theory they are taught during the courses into practice. They work in groups of 3-8 people over a period of four months. The theory and the process, as well as the final product, have to be documented in a report, which is usually between 80 and 120 pages long. The major part of this report is written during the last three weeks of the project period.

The students experienced problems, not so much during the programming process where existing tools seem to be of sufficient help, as during the writing process which is usually short and hectic and characterised by a very dynamic organisation of tasks and responsibilities. They especially had problems in keeping an overview of the document and how its structure develops through new versions. This caused them to have problems in establishing baselines of the document, to track structural changes and to find proper use of version histories. Finally, communication of information about the development is important as these students often work in a distributed way. Some students in a group may work from home, while others work from the room that each group has at the university, and others yet work from the computer labs at the department. This fact led to creating an environment called CoEd [BLNP97].

The problems of these students are typical examples of the problems present in collaborative writing efforts. Instead of buying a new tool or trying to solve the general problem, our strategy was to try to solve the specific problems of the students in their given context, and to gain experience from the students actually using a prototype of our tool.

All groups of students use Latex for producing their project reports. Some groups have so far used self-imposed group discipline to be able to manage the development, dividing the document up into disjoint parts with respect to responsibility. They have, however, usually encountered serious problems, both because parts of the report are inherently interdependent and because of the complete absence of versioning of the compound document. Most groups have used either RCS [Tichy85] or CVS [Berliner90] as their tool of choice to manage the development, usually based on whether they liked a strict locking mechanism or not. This enabled them to version the development of the single parts of the document, but they still had problems in keeping an overview of the entire document and in manipulating its structure.

2.1 The Problems in Collaboration

The work on CoEd [BLNP98] took its origin in the problems that students had reported from their co-operative work on developing textual documents. The problems experienced were many and varied, but can roughly be grouped into three categories. One that has to do with the lack of overview and co-ordination, both of the document

and of what everyone else is doing. Another category that has to do with problems doing version control and change tracking the way that they want and need. And, finally, there are problems that have to do with the communication of information.

Problems that have to do with the lack of overview and co-ordination manifest themselves in several ways. It is very difficult to organise the structure of the report and to have the structure visualised while working in front of the screen. As a result, indices or entire reports are printed on paper to gain overview and much work is lost in manually changing Latex commands (and/or file names) to reflect a reorganisation of the report. This also implies that groups rarely change their way of working. If they work in a top-down fashion, the structure of the paper remains fixed right from the start. Groups working in a bottom-up way remain in a limbo until the very last moment where finally all the pieces can be put together.

These problems are, in part, due to the fact that if we divide the document up into several files, reflecting its hierarchical structure, then the version control tool is treating those just as single pieces and not as a whole too. In part, the problems are due to the lack of a proper GUI that can visualise the structure of the document. Version control tools permit us to divide the document in logic entities, like chapters and sections, reflecting also the division of responsibilities. However, without a proper GUI it is difficult to get a quick overview of the entire document. Furthermore, the fact that the structure is only implied by a directory structure, means that we must manually change this structure every time the organisation of the document is to be changed.

To remedy this problem the CoEd system has knowledge of Latex, so it can automatically create (and maintain) the storage organisation from that implied by the Latex code. Furthermore, the GUI is capable of visualising documents using the Latex structures. Finally, it is possible to visualise - and work with – both the document as a whole and its individual parts.

In the second category of problems we find misfits between the version control needs of the students and the functionality provided by the tools they use. Their needs for version control are not very sophisticated. They do not develop variants and do not have to maintain old versions, as it is usual in software development. Still they have troubles in finding help from traditional version control tools. They have problems identifying and retrieving old versions. Often confusion arises when the supervisor comments on a document and the students find out that it is not the version they printed out just before the meeting. When a section or a chapter is split into two, the version history for one of the parts is lost. These problems are similar to the problems in version selection, baselining and change tracking pointed out in [Tichy88].

Again the problems are, in part, due to the lack of a GUI, and, in part, to problems with the data model that the version tools build on. An adequate GUI makes version selection far easier because one immediately sees what one selects, at least with versions of the individual parts. Baselining the entire document is a cumbersome and sometimes error prone process. This is due to the fact that it is a manual task where the document is viewed as a collection of versioned parts. As such, there is no explicit versioning of the collection as a whole. Furthermore, as the tools are unaware of

operations like splitting a unit, this becomes something that is unsupported and has to be carried out outside of the tool's control.

To avoid these problems we made the versioning of a document's structure an integral part of the tool, treated on equal footing with the versioning of the individual parts. Furthermore, we supported splitting of units as a basic functionality. Finally, the GUI facilitates identification and selection, and visualises the result immediately.

2.2 The Architecture of CoEd

The architecture of CoEd is built around the principle that a Latex document has a hierarchical structure and as such consists of a set of leaves and internal nodes, each of which can contain text. Leaves and nodes are the smallest granularity of the system and are called *units*. For the versioning of a unit we use the traditional approach of creating a version group for the unit and let the development of versions be reflected by a version graph. The root node has a special status as it represents the whole document. The root node is versioned just like all other nodes and this provides us with versioning of the document as a whole. A given version of the whole document is called a bound configuration in accordance with the terminology used in software configuration management.

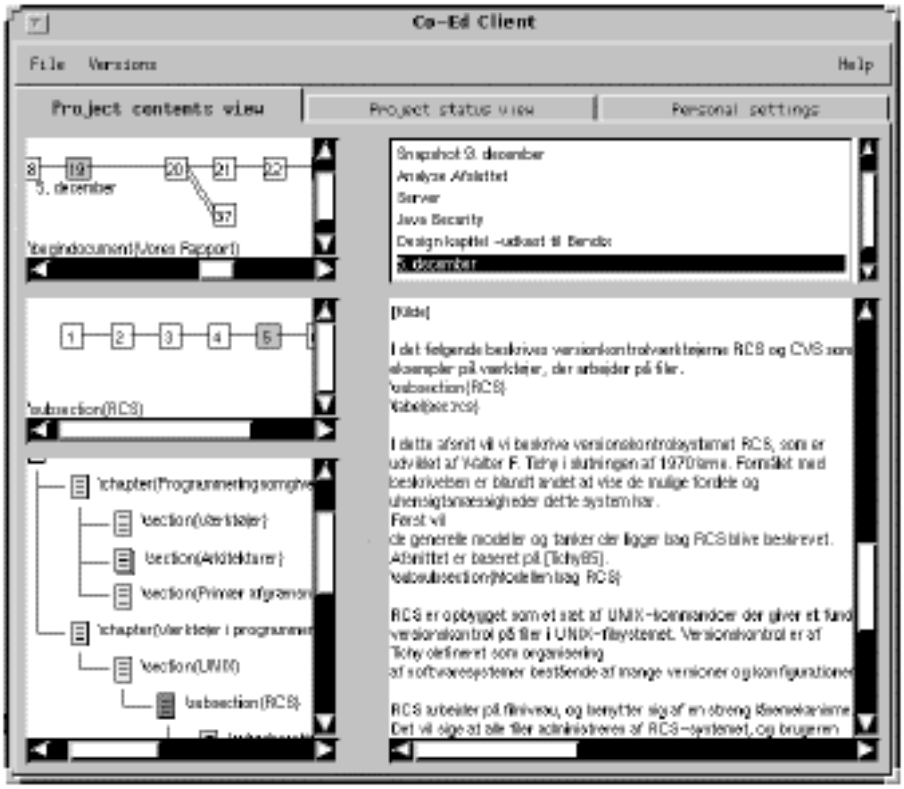

Figure 1. How CoEd presents itself to the user.

CoEd has four browsers which each shows a different aspect of the internal data structure. This makes it possible to look at the data structure (i.e. the document) at varying levels of details allowing for a flexible granularity. Figure 1 shows the GUI of the CoEd system.

The *hierarchy browser* is found at the bottom left. It shows the hierarchical structure of the document as it is implied by the Latex commands. Icons can be expanded and compressed by double-clicking them. This makes it easy to get a quick overview of the document at the desired level of detail. We find the *text browser* at the bottom right. Here is shown a contiguous piece of text that corresponds to the icon selected in the hierarchy browser and the text that immediately precedes or follows the selection.

At the top left, we find the *version browser*, which consists of two windows. The top window shows all the versions of the bound configuration, which is equivalent to showing the version group of the root. In the bottom window is shown the version graph for the unit that is selected in the hierarchy browser.

At the top right, we find the *baseline browser*. This browser was introduced in order to solve the lack of overview caused by the high number of bound configurations. As can be seen in the version browser window, bound configurations can be named to distinguish important ones. These named bound configurations are the ones that appear in the baseline browser. The selection of one of these baselines will cause the

icon of the corresponding bound configuration to be selected and highlighted in the version browser.

From figure 1, we can see that initially the user selected the "5. December" baseline in the baseline browser. This caused CoEd to find and select that version of the document (version 19) and highlight it in the upper window of the version browser. CoEd also finds and displays the structure of this bound configuration version in the hierarchy browser. Then the user selected "\subsection{RCS}" as the unit he is interested in and CoEd found and displayed the text of this unit (and of immediately surrounding units to fill out the text window) in the text browser, highlighting it. CoEd also displayed the version group for the selected unit (version 5) in the lower window of the version browser.

2.3 How CoEd Is Used

A typical scenario for the use of CoEd will find us starting with (a piece of) a document which we now want to continue to develop using CoEd. Using the file menu, we will ask CoEd to check in the file containing the document. CoEd parses the Latex code and, if successful, constructs the implied hierarchical structure, otherwise it refuses the text.

Using the version browser, we now select the bound configuration we want to change (usually the latest). We can now use either the hierarchy browser or the text browser to select the contiguous piece of text we want to change (it can span several logic units) and ask CoEd to check it out to a single file.

This file we can edit using our favourite editor and when we have finished editing the text, we ask CoEd to check it in again. CoEd automatically discovers which units have been changed - creating new versions - and which have not - leaving them untouched. It will even discover if units have been added or deleted and react correspondingly.

2.4 Advanced Functionality in CoEd

CoEd also has some more advanced functions that work at the structural level of the document. These are split of a unit, creation of meta-versions and direct manipulation of the structure.

The students create many versions of their documents during the writing phase. Because we consider the document as a whole, each change to one of its parts means that a new version of the whole document is created. The baseline browser reduces this high number of bound configurations, such that it becomes manageable. If changes to more than one unit is carried out in the same edit session, this will create more than one version of the document. This is not a consequence of the extensional versioning used at the document level, but due to the lack of a session concept in CoEd. Meta-versions were introduced to automatically group together all the versions created in one single edit session. Meta-versions can be opened such that the single versions in the meta-version can be accessed.

Let us assume that we have chapters A, B and C, and want to split chapter A into two chapters A1 and A2. When we check in the result - A1, A2, B and C - CoEd will discover that there is one more chapter than was checked out. It will, however, also recognise that A1 and A2 were parts of the original chapter A and create two new version groups and connect them with the version group of the original A in a seamless way, in order not to loose continuity in the compound version history.

It is also possible to directly manipulate the structure and in this way permute units. It is possible to move both single units and parts of the structure. We simply select what has to be moved and then drag it to the place where it has to be inserted. In this way, we can change a chapter to a section (including its sub-structure) or vice versa, and CoEd will make the necessary changes to the Latex code for us.

2.5 Experience with CoEd

We have implemented a working prototype of CoEd and students at Aalborg University have used it for developing their project reports. The results from these experiments are rather promising. The average number of pages handled by CoEd was about 80 pages per project. Some groups just used it to play around because they did not trust the stability of CoEd and were afraid of loosing their data. These groups had relatively little text (about 50 pages), while a few groups used it seriously and had about 120 pages under the control of CoEd. In the end, CoEd turned out to be amazingly stable and very little data was lost the few times it crashed.

The average number of units per project was 147, again with serious user going higher. The number of bound configurations was 2 to 6 times the number of units. That the students were able to maintain an overview anyway, proves the value of the baseline browser and the meta-version concept. There were about 9 versions in meta-versions, but up to 20-30 versions were seen. Especially for groups that brought larger pre-written pieces of text into CoEd rather than using CoEd from the very beginning.

Development was mostly linear with very few branches and merges. Split of units was used but not extensively. The direct manipulation of the structure, on the other hand, was used very extensively and was rated by the students as one of the strongest points about CoEd. And, above all, the fact that direct manipulation could be carried out under full version control and therefore could be undone very easily.

Students also felt that this kind of version control and change tracking lowered the need for face-to-face meetings for exchanging information. This indicates that in the past many such meetings were held mainly for communicating information and for coordination purposes, and that by using CoEd they were able to reduce the needs for coordination.

The improved support for handling entire documents led to the discovery of new functionality that was desired. The students asked for better visualisation of differences in structure between versions of documents. They wanted to be able to compare parallel versions, both at the structural and the textual level. And, finally, they wanted to be able to attach more information to versions of documents and units. These exten-

sions were difficult to carry out in the current implementation of CoEd. This, and the fact that fine granularity and versioning of structures is not well supported by traditional version engines like RCS, caused us to look for a version engine that was better suited to the requirements of CoEd for supporting change tracking.

3 Change Tracking

VTML is a descriptive data format for fine-grained change tracking. It is not a versioning *system*, but a flexible data format that can be used by systems that implement a wide range of versioning styles. It was born from the tentative of determining an adequate versioning style for hypermedia documents in a collaborative environment ([DHHV94] and [HHDV95]). The versioning styles allowed by VTML can vary from extremely informal and unstructured asynchronous collaboration patterns among creative writers, to the formalised and controlled sequential actions of a team of programmers, to the synchronous access to a shared blueprint by a team of architects and designers. It can be flexibly used with a large number of system architectures, varying from flexible editing clients and dumb storage servers, to extremely dumb clients interacting with a sophisticated versioning server.

The format is designed to be consumed by programs, and so it is relatively terse and simple to parse. Although we are currently applying VTML to the management of text, any text or binary format can be directly represented in a VTML document. In particular, VTML was designed initially to handle the management of versions for HTML documents [VD95], and in general for managing versions of all kinds of markup languages (such as SGML, XML and all other derived languages).

VTML-based systems may make use of the features of VTML to obtain a few interesting features, such as:

- the version history may branch, creating a tree of variants. The version history may also converge, creating a master version that inherits from several different variants by some form of user-guided or automatic merge.
- locks to control accesses to authors are not necessary. This is a consequence of allowing branching versions: conflicting check-in operations can always be allowed, automatically creating new branches of the version tree if necessary. The versions can then be “harmonised” with a merge operation.
- a check out operation is not necessary: users may use copies without synchronisation control by a server or a distributed consistency algorithm, for instance by using a local copy on the client’s file space.
- full sequential undo can be easily provided. Since each operation is logged and identified, it is easy to rebuild the state of the document at any point in time. In Palimpsest [Durand93], VTML is even used to allow arbitrary undo: by explicitly expressing the existence context of each change, one can accept or reject arbitrary past operation regardless of their sequence in time.
- automatic version identification is supported, according to a series of numbering schemata. Four numbering schemata can be used, each having equivalent expressive power.

- VTML versions provide a consistent and reliable addressing mechanism for document spans, that requires no modification to the document and can survive unmonitored changes to the document itself. One important service that a VTML-based versioning system can provide is to precisely locate the position of data designated by an offset into a previous version of the document. This is an important operation for the support of external link bases that refer to changing data. The same mechanism can also be used to provide flexible document fragment re-use, with little additional machinery.

VTML stores information about all single modifications to the shared document. It is able to report that something as simple as an insertion has taken place, or something as complex as a sort. Since the list of possible operations is open, VTML describes every complex change as a list of simple operations: insertions, deletions and modifications. Thus the basic purpose of the language (i.e., to be able to build a given version of a document according to the changes it has incurred into since its creation) is preserved even if the meaning of the actual operations is unknown.

Attributes are associated to single changes. This allows an extreme flexibility in describing them. In order to avoid overloading of repeated data, shorthand facilities are provided. The list of data items that can be associated to every change is also open, and possibly very large. Thus, instead of listing extensively the kind of attributes, only a few necessary ones are determined, and a way to add new ones is provided. The necessary attributes are basically used to univocally determine the whereabouts and the correct grouping of the changes. Everything else, from the author or the date of the change to the comments about a given change, or to the author's shoe size, for that matters, is an additional attribute that is not part of the language.

VTML comes in two equivalent formats: the internal format stores side by side the modifications in the positions they have happened. The external format stores them in the chronological order they have happened. A VTML document is composed of one or more VTML blocks, contained within a {VTML} {/VTML} set of tags. VTML blocks are composed either of internal markup (using the elements ATT, USROP, INS, and DEL) or external markup (using the elements ATT, USROP, EXTINS, EXTDEL). The same document may contain VTML blocks of both types.

All change commands that are described with internal tags are stored within a single VTML block, while external tags may be stored in as many blocks as needed. Applications that require support for both internal and external changes in a single file may concatenate multiple blocks together.

VTML is meant to provide change tracking support to markup languages such as HTML and XML. HTML 4.0 [RLJ98] includes two new tags, INS and DEL, that are meant to express changes from previous versions of the same document (e.g., in legal texts); On the contrary, VTML lies on a completely different layer, having the markup as content of its tags. This is due to several reasons:

Different handling of tags and content: users don’t simply add content during edits, but may modify tags and attributes. If versioning tags are at the same level as content tags, and parsed at the same time, it is impossible for them to keep information about modifications of content tags. The duplication of tags, and the need for

the versioning tags and the document tags to create a legal (extended) HTML document also creates problems with ensuring proper nesting of tags.

Potential for misuses, hacking, and manual modifications: if the document is edited by a versioning-unaware editor, or, even worse, manually, versioning information will inevitably become outdated, inconsistent and possibly will generate a corrupt HTML document.

Complexity of resulting document: HTML is an SGML DTD thought for simple content markup, hypertext links and the like. Versioning tags do *not* modify the appearance or role of the parts of the document but perform a more pragmatic and low-level chore: helping determine whether a piece of the document belongs to a specific version or not. Mixing semantic and rendering markup with content-determination tags creates extremely complex and unreadable documents.

In summary, adding special tags to a markup language does not help change-tracking: being part of the markup language, these tags cannot express changes in the markup itself or changes that disrupt the correct nesting of the markup (e.g., two paragraphs that have been joined, or a link destination that has been changed). The only solution we find acceptable is to foresee two independent parsing steps, the first of which considers and activates just the change-tracking information, building a complete marked-up document, and the second that parses the specific document markup and creates its visual representation.

3.1 A Complete Example Using VTML

Basically, VTML tags are meant to describe the editing operation performed on the document, and describe operations that are not the result of changes in the document data, but rather the selection of some existing changes. Let us suppose we have the following situation: David and Lars are collaborating on writing a document.

First version: Lars inserts the string: “The quick brown fox jumps over the lazy dog.”

Second version: David substitutes “quick” with “speedy”, and removes “lazy“: “The speedy brown fox jumps over the dog.”

Third version: Lars substitutes “brown” with “red” and inserts “sleepy” before “dog”: The speedy red fox jumps over the sleepy dog”.

For reasons known to the VTML engine, version 1 and 2 are stored together with the internal markup, while version 3 is stored externally (maybe the engine hasn’t had the time yet to import the new version). Versions 1 and 2 correspond to the following VTML block:

```
{VTML NAME=“Hunting” CVERS=2 _AUTHORS=“Lars, David”}
{ATTR ID=1 vers=1 _author=“Lars”}
{ATTR ID=2 vers=2 _author=“David”}
```

```
{INS ATT=1} The {INS ATT=2} speedy {/INS} {DEL ATT=2}
quick {/DEL} brown fox jumps over the {DEL ATT=2} lazy
{/DEL} dog. {/INS}{/VTML}
```

Each VTML tag describes the shared context given, at least, by the document name, and the current version, and, in this case, also by the group of legal authors. The ATTR tag stores a few attributes that should be repeated several times in the document tags, and that are associated with the ATT attribute of the actual tags. Therefore, writing

```
{ATTR ID=1 vers=1 _author="Lars"}
{ATTR ID=2 vers=2 _author="David"}
{INS ATT=1} The {INS ATT=2} speedy {/INS} {DEL ATT=2}
quick {/DEL} brown fox jumps over the {DEL ATT=2} lazy
{/DEL} dog. {/INS}
```

is equivalent to writing:

```
{INS vers=1 _author="Lars"} The {INS vers=2
_author="David"} speedy {/INS} {DEL vers=2
_author="David"} quick {/DEL} brown fox jumps over the
{DEL vers=2 _author="David"} lazy {/DEL} dog. {/INS}
```

INS and DEL represent the actual changes that were performed on the text. Since the text is a linear sequential format, there is no need for modification operations, but we can safely restrict to insertions (INS tags) and deletions (DEL tags).

On the other hand, this is an external representation of version 3:

```
{VTML NAME="Hunting" CVERS=3 _AUTHORS="Lars, David"}
{ATTR ID=1 SOURCE="Hunting" VERS=3 _author="Lars"}
 {EXTDEL ATT=1 POS=15 LENGTH=5}
 {EXTINS ATT=1 POS=15}red{/EXTINS}
 {EXTINS ATT=1 POS=42}sleepy {/EXTINS}{/VTML}
```

This VTML block contains an external description of the changes leading to version 3. In this case, insertions (stored as EXTINS tags) specify their position, while deletions (EXTDEL tags) specify both their position and the number of removed characters.

Separately, Fabio opened version 2 of the "Hunting" document and made some other modifications: he substituted “jumps over” with “is not caught by” and inserts “Today” at the beginning of the sentence: “Today the speedy brown fox is not caught by the dog.” Therefore, the following is the result of his modifications:

```
{VTML NAME="Hunting" CVERS=3 _author="Fabio,Lars,David"}
{ATTR ID=1 SOURCE="Hunting" VERS=3 _author="Fabio"}
```

```
{USROP ATT=1 REF=2 NAME="SUBSTITUTION"}
 {EXTDEL POS=29 LENGTH=10}jumps over{/EXTDEL}
 {EXTINS POS=29}is not caught by{/EXTINS}{/USROP}
{EXTINS ATT=1 POS=1}oday t{/EXTINS}{/VTML}
```

This version, besides making use of the external representation of changes, uses the USROP command, which collects into a single operation a sequence of basic editing commands (insertions, deletions, modifications). In the external format, the USROP tag groups together the basic operations it is composed of, and labels them with a human-understandable name.

The first problem with accepting this version is that both versions claim to be version 3, since both were created from version 2 in absence of other derived versions. Lars decides that his own version will remain in the main branch of the version tree. This affects the numbering of the versions, as Fabio's version is renumbered and becomes 3.1. Then Lars merges Fabio's contributions into a new version: he accepts the substitution of the verb, but NOT the insertion of "Today".

This is a structure of the version tree:

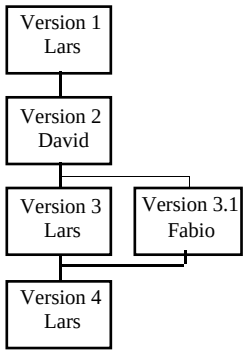

Figure 2. The version tree of the VTML example.

The engine easily generates the following internal representation:

```
{VTML NAME="Hunting" CVERS=CURRENT AUTHORS="Lars,David,
Fabio"}
{ATTR ID=1 ref=1 vers=1 _author="Lars"}
{ATTR ID=2 ref=2 vers=2 _author="David"}
{ATTR ID=3 ref=3 vers=3 _author="Lars"}
{ATTR ID=4 ref=4 vers=3.1 _author="Fabio"}
{ATTR ID=5 vers=CURRENT _author="Lars"}
{USROP ATT=4 NAME="Substitution" REF=6 INCLUDES="5"}
{USROP ATT=5 NAME="Merge" EXCLUDES="7"}
{INS ATT=1} T{INS ATT=4 REF=7}oday t{/INS}he {INS ATT=2}
speedy {/INS} {DEL ATT=2} quick {/DEL} {DEL ATT=3} brown
{/DEL} {INS ATT=3} red {/INS} fox {DEL REF=5} jumps over
```

```
the {/DEL} {INS REF=5}is not caught by {/INS} {INS ATT=3}
sleepy {/INS} {DEL ATT=2} lazy {/DEL} dog. {/INS}{/VTML}
```

The main features of this version are that the internal form of USROP has been used and that a merge has been performed. The internal format of the USROP tag specifies the basic operations it is composed of by listing their REF number in an INCLUDES attribute or listing the other ones in an EXCLUDES attribute. A merge is just another USROP operation where the relevant operations are either accepted or ignored in the merged version. Thus, in this case, version 4 is composed of a single operation that merges all previous operations except for the one with REF = 7.

This new block can either be stored as such by the VTML engine or divided again into elements and stored separately. When the engine saves the document, it will substitute the CURRENT value with the appropriate version number (in this case, 4).

4 Integrating CoEd and VTML

In this section we briefly describe the reasons for integrating CoEd and VTML, the interface between the two systems, and a few example scenarios where using VTML can provide additional functionality to the CoEd collaborative system.

CoEd has proven itself a strong and flexible tool to use for supporting collaborative writing through change tracking, versioning of whole documents and management of document structure. We have, however, found some things to improve through our experiments with the prototype. While CoEd's interface and model layers (see figure 3) work rather well, the engine layer is far too simple, since it does basically nothing but system calls to the file system. CoEd stores each version of a unit in its entirety and does not even try to use space-saving delta mechanisms. This means that using CoEd becomes prohibitive in a larger scale as it really burns up disc space. We made this initial choice because we wanted to put emphasis on the concepts and development of an experimental prototype rather than on an efficient implementation.

In order to further develop the functionality of CoEd and to make it a more efficient tool that can be used for real projects, a more powerful and flexible engine is needed. We have looked into traditional tools for version control, like RCS [Tichy85], and we found that they are simply not powerful enough. Such tools are very efficient in representing version groups in as little space as possible, but they are limited in that they do not go beyond version groups. This still leaves the versioning model of CoEd the task of managing the structure of the document and of versioning this structure.

VTML efficiently represents changes in a versioned text, so that VTML-aware applications may make use of the change-tracking facilities of the language to provide sophisticated versioning support to its users: version selection, branching, comparison, and merge. VTML therefore seems like an optimal choice for the engine component of the next CoEd prototype since it can handle and version both contents and structure. VTML is a language, not a tool, so we had to decide what kind of VTML-enabled application we were looking for. A VTML engine can provide basic parsing

and storage functionality. By adding a simple interface layer for the CoEd applications, we can easily provide sophisticated versioning functionality.

In figure 3, we show the overall architecture of the foreseen application.

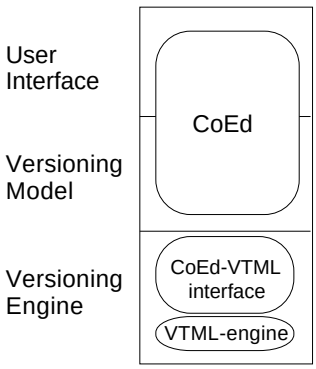

Figure 3. The conceptual architecture of CoEd with VTML.

The interface layer provides the operational interface between CoEd and the VTML engine, and consists of the following operations:

```
Put_version(data:Data_structure,
      depends_on:Version_name) ->
      Version_name;
```

The Put_version operation appends a new version to an existing document. This corresponds to a check-in operation for the VTML engine that generates a diff between the specified version name and the new one submitted. Based on that, the engine determines the VTML coding and the version number corresponding to the new version. The VTML engine will then decide autonomously whether to store the new version using the external format in an autonomous file, or to insert it as internal coding in the existing one. Finally, it will return the new version name for CoEd to update its internal database.

```
Get_version(version:Version_name) -> Data_structure;
```

The Get_version operation creates the required version. This corresponds to a check-out operation for the VTML engine. The engine will retrieve all the versions leading up to the requested one, and will perform the change operations stored in them necessary to build the requested version. It will then return the data corresponding to the requested version.

```
Compare_versions(versions:Version_group_list,
      deleted_data: Boolean) ->
      Comparison_data_structure;
```

A comparison structure is simply a text document that contains some colour coding information. The CoEd model will request a list of versions to be displayed together to ease the comparison. For each version, it will suggest a colour. In the deleted_data parameter, it will then specify whether deleted data should be displayed or not. This corresponds for the VTML to a multiple check-out operation where instead of simply building the requested versions, each version is assigned a colour coding that will be used to specify the display of each document bit. If deleted data are requested, the deletion operations are not performed, but the corresponding data are left in the document with an additional special colour coding.

To clarify the working of the CoEd+VTML system, we examine four possible scenarios where the system is used and provides sophisticated collaborative functionality:

I - Creating a new document

Student A places a sharable and existing document under the wings of CoEd.

In this case CoEd will parse the text of the document and create the hierarchical structure implied by the Latex commands. For each of the leaves and internal nodes in this tree, it will create a new version group and insert the text of the unit as a first version in this version group.

II - Getting and modifying a document

Student B makes a modification to the document's latest bound configuration and saves it.

When the text is checked in, CoEd discovers which parts have been modified. For all the modified units it calls the VTML engine to have new versions created and stored.

VTML handles and stores each single change that has happened to a document between saves. This means that, after each editing session, the VTML engine must determine what has changed since the last saved version. Since there are presently no plans of integrating a VTML-aware editor into CoEd, the difference is determined by making a diff of the two versions. The output of the diff program is then converted into VTML commands, and passed back to the VTML engine. The VTML engine now can choose between using the internal format, and creating a single VTML file containing all the existing versions of the document, and using the external format, which can then be stored independently of the rest of the document, in an autonomous file. The choice is done according to reasons of efficiency and availability of the new version.

III - Comparing different versions

Student C accesses student B's bound configuration and wants to compare it with a previous bound configuration.

The CoEd interface transforms this command in a request to the VTML engine for two different versions of the document. The VTML engine verifies whether those versions of the document are stored externally. In this case internalises them and generates the compact internal representation of the selected versions of the document. Then it transforms the relevant change instructions in colour choices for the text of the document display, thereby allowing the comparison of the two versions. This is simply done by eliminating version information for those bits that belong to both version, and converting the version information into colour instructions for those bits that have been modified in either version.

This information is then visualised in a separate window by the CoEd GUI.

IV - Parallel access to a document

Students A and C want to make modifications to the same bound configuration at the same time.

Any check out of text in CoEd is done within the context of a bound configuration. It is possible to make more than one check out from the same bound configuration – either in parallel or sequentially. CoEd notices that a branch has to be created and handles it at both the structural and the textual level. The structural level is handled internally, while managing parallel variants of text is handled by the VTML engine.

Since the VTML engine easily allows branching, neither student is blocked from accessing in write mode the document. We are not planning to use VTML-aware editors or notification mechanisms, so at save time the two versions are autonomously accepted by the VTML engine and put in two parallel variants. Since VTML allows parallel variants to coexist without requiring to merge the incompatibilities, and since VTML is able to provide any selection of versions even if belonging to different version branches, there is no pressure to resolve the inconsistencies that may have been created during the parallel edits. Once the need to harmonise the differences becomes paramount, the merge operation can be activated from the CoEd GUI. A merge can either be done automatically or manually. In both cases a person or an algorithm will select, for each edit that appears in either relevant branch, whether it should belong to the final version or not. The merge version therefore is an optional operation that reconciles different version branches of the same document without loosing information on each composing branch.

5 Conclusions

The CoEd environment has proven to be robust and useful in many collaborative situations. On the other hand, the simplicity of the underlying storage engine has prevented interesting functionality to be added.

It was an important decision to maintain CoEd interface characteristics and versioning policies, and improve on the underlying storage and composition mechanisms. VTML provides the sophistication needed in the management of the versions,

and thus allow to improve the feature set of CoEd. Furthermore, the significant space savings available with the VTML format may easily make CoEd usable in heavily real-life situations.

CoEd is currently a monolithic working environment. Work is in progress to move it to a client-server architecture. The WebDAV extensions to HTTP for distributed authoring ([SVWD98] and [GWF*]) would help by providing a standard way for clients and servers to interoperate. The Delta-V working group, derived from the WebDAV working group, is currently establishing the extensions to HTTP needed for versioning and configuration management. Within both groups, VTML has been proposed and extensively discussed. Unfortunately, the consensus within the working group has been that change management operations, being media-dependent, are out of scope, and will not be covered by the forthcoming standards [AC99].

Acknowledgement

This work has been supported, in part, by the Danish Research Council, grant no. 9701406.

References

[AC99]: J. Amsden, G. Clemm: Web Versioning Model, *INTERNET DRAFT, draft-ietf-webdav-versionmodel-00.html*, in the version of February 1, 1999 expiring August, 1999. http://www.ics.uci.edu/pub/ietf/deltav/model/model990209/

[BLNP97]: L. Bendix, P. Larsen, A. Nielsen, J. Petersen: CoEd - A Tool for Cooperative Development of Hierarchical Documents, Technical Report R-97-5012, Department of Computer Science, Aalborg University, Denmark, September 1997.

[BLNP98]: L. Bendix, P. Larsen, A. Nielsen, J. Petersen: CoEd - A Tool for Versioning of Hierarchical Documents, in *Proceedings of SCM-8* (Bruxelles, Belgium, July 1998), Lecture Notes of Computer Science, Springer Verlag.

[Berliner90]: B. Berliner: CVS II: Parallelizing Software Development, in Proceedings of USENIX Winter 1990 (Washington, DC, 1990).

[Durand93]: D. Durand: Cooperative Editing without Synchronization, in *Hypertext '93 Workshop on Hyperbase Systems* (Seattle, WA), Technical Report n. TAMU-HRL 93-009, Hypertext Research Lab, Texas A&M University, College Station TX

[DHHV94]: D. Durand, A. Haake, D. Hicks, F. Vitali (eds.): *Proceedings of the Workshop on Versioning in Hypertext Systems*, held in connection with The European Conference on Hypertext, ECHT94. Available as GMD Arbeitspapiere 894, GMD - IPSI, Dolivostrasse 15, 64293 Darmstadt, Germany.

[GWF*]: Y. Goland, J. Whitehead, A. Faizi, S. Carter, D. Jensen: HTTP Extensions for Distributed Authoring - WEBDav, *Internet Informational Request for Comments(IETF RFC) 2518*, ftp://www.ietf.org/rfc/rfc2518.txt

[HHDV95]: D. Hicks, A. Haake, D. Durand, F. Vitali (eds.): *Proceedings of the ECSCW'95 Workshop on the Role of Version Control in CSCW Applications*, available as http://www.cs.bu.edu/techreports/96-009-ecscw95-proceedings/Book/proceedings_txt.html,Boston University Technical Report 96-009.

[MA96]: B. Magnusson, U. Asklund: Fine Grained Version Control of Configurations in COOP/Orm, in *Proceedings of SCM-6* (Berlin, Germany, March 1996), Lecture Notes of Computer Science, Springer Verlag.

[RLJ98]: D. Ragget, A. Le Hors, I. Jacobs: *HTML 4.0 Specification*, W3C Recommendation, http://www.w3.org/TR/PR-html40/

[SVWD98]: J. Slein, F. Vitali, E. J. Whitehead, Jr., D. Durand: Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web, *Internet Informational Request for Comments(IETF RFC) 2291*. February, 1998. Also as *ACM StandardView* 1 (5), 1997, p. 17-24. ftp://www.ietf.org/rfc/rfc2291.txt

[Tichy85]: Tichy, W. F. RCS - A System for Version Control, Software - Practice and Experience, Vol. 15 (7), July 1985.

[Tichy88]: Tichy, W. F. Tools for Software Configuration Management, in *Proceedings of SCM-1* (Grassau, Germany, January 1988).

[VD95]: Vitali F., Durand D., Using versioning to support collaboration on the WWW, in *The World Wide Web Journal*, 1(1), O'Reilly, 1995.