Fine Grained Version Control of
Configurations in
COOP/Orm

Boris Magnusson
Ulf Asklund

LU-CS-TR:96-166

Also published in: Proceedings of SCM6, Symposium on Configuration
Management, I. Sommerville (Ed.), Berlin, March 1996,
LNCS, Springer Verlag

Department of Computer Science
Lund Institute of Technology
Lund University

P.O. Box 118, S-221 00 Lund
Sweden

Fine Grained Version Control of Configurations in COOP/Orm

Boris Magnusson and Ulf Asklund

Dept. of Computer Science, Lund Institute of Technology,
Box 118, S-221 00 Lund, Sweden
E-mail: {Boris | Ulf}@dna.lth.se

Abstract. This paper describes a unified approach to version control of documents and configurations. Hierarchical structure, which is present in most documents such as programs, is recognized and utilized in a fine-grained version control system. The same mechanism is used for version control of configurations and extended to handle DAGs as well as trees. Change propagation within one hierarchical document is automatic while bindings between documents are explicit. The model is novel because of its integration of version and configuration control, fine-grained version control, and explicit graphical user interface. It supports teams of distributed users by offering optimistic check-out with strong support for merging of alternatives.

1 Introduction

Software systems are made up from hierarchical collections of hierarchical documents. Traditionally, version control has been applied to keep track of the revisions of individual documents, while configuration management has focused on how to form systems or sub-systems out of collections of documents [Roe75, SCCS, Tic85, Tic88]. Although this separation has some benefits in factoring out minimal functionality into single tools it suffers from the lack of integration. We will here illustrate some of the most severe problems with this approach as we see it, without any ambition of making the list complete.

Document size There are conflicting demands on the size of the involved documents. Even a small change to a document creates a new version of the whole document. From a version control point of view it is a benefit if documents are kept small since the precision of the information the version control system will give us will get higher.
From the configuration management point of view it is an advantage if the documents are fewer (and thus larger) since the complexity grows with the number of documents involved and with their versions. The number of meaningless or non-compatible configurations of versions of documents grows exponentially.

Change Size It is often the case that a change affects only a small part of a document. Still, the version control and locking scheme is based on the whole document which is often found as unnecessary coarse.

Related documents It is often the case that many documents are tightly related and are in fact version controlled together, but many systems can not represent the connection between related changes to different documents.

Concurrency control Lock on check-out, as commonly used by many version control systems, gets awkward to use when the group of people involved grows. With a locking system there is a drive for using many small documents since then more people can work

simultaneously without needing to change the same piece of information at the same time. Locking also makes such a system hard to use in a distributed environment.

Configurations Configurations are often only described indirectly through make-files, and although these can be versioned they can not handle structural changes to a configuration since the underlying file system is not versioned.

Awareness It is often hard to find out what documents other developers have changed, and what changes they have made, or even who checked out a particular document. Providing some level of awareness also seems essential in providing flexible work process support.

Some recent systems have identified and addressed some of the problems above. As an example, CVS[Wat,Ced93] can manage collections of files and also allow multiple checkouts, but give only rudimentary help for merge. TeamWare [Team] facilitates distributed development by allowing replicated repositories and facilitating merge of the (SCCS) history files. Again with weak support for merge of the documents themselves. ClearCase [Cla95], on the other hand, has strong support for merge, resolving simple differences automatically and identifying conflicts for human resolution. This through an interface that, according to Ovum [RBI95] sets a new standard for the industry.

To the best of our knowledge no existing configuration management system addresses the problem of awareness.

Version control can be seen as added to current file systems as an afterthought and editors, compilers etc. do not deal with this information. We see version control as essential for any system development and it should therefore be a basic mechanism understood by all processors.

1.1 Our approach

The starting point for this work has been the aim to support teams of programmers working together, providing a collaborative editing environment, an area that combines problems from both CSCW (Computer Supported Cooperative Work) and SE (Software Engineering). Collaboration and sharing information naturally demands a version control system and ambitions to support also synchronous editing [MM93] has led us to support unusually fine-grained versions. We have also taken the position that systems that use a pessimistic, lock on check-out, approach do not scale up to many users in a geographically distributed environment. This has lead us to design our environment on an optimistic approach where developers always can create new versions (forming a new alternative if necessary) and then providing strong support for merging alternatives using an operation-based diffing approach [LvO92]. For configuration management we have taken the position that configurations should be unified as much as possible with other documents and for example also be version controlled with operation-based diffs [MAM93].

Another starting point for this work is the Orm environment [MHM+90] developed in the Mjølner project [KLMM93]. This is a tightly integrated environment built on incremental techniques. Here there are no processors like compilers and linkers visible to the developers, this is merely functionality offered, and managed by the environment. The developer edits and executes. The integrated approach chosen in constructing Orm has also included a storage model for programs in structured form [Gus90]. This storage form has many similarities with engineering databases as described in [Kat90]. The Mjølner/Orm environment includes version and configuration facilities. Compared to what is presented in this paper it is working on a much coarser level and offers a less user

friendly interface. Nevertheless it is offering versioned connections between modules and also to ‘grammars’ defining the language implementation. It is the experience from this environment that have lead to the further development presented in this paper. ‘Fine grained version control’ as a term was introduced in Orwell [TJ88] meaning version control at the method level rather than at the module or class level in object oriented programming. We use the term in the same meaning, but go further in decreasing grain size. We record each edit operation using ‘operation based diffs’ in the meaning of [LvO92] and encourage creation of versions more frequently, and thus extend the meaning of fine-grained version control.

Our research has been guided by a goal to explore the possibilities to increase the level of functionality offered to the users. In doing so we have chosen an integrated approach for our prototype environment and put less emphasis on how the parts of the environment integrate with existing systems. As an example we do not put priority in this phase of our work on how to make it possible to use existing editors (such as emacs, which are not version aware) with our system. The situation can be compared with introducing word processors which also represent integrated environments where existing text editors can not be used. Although not everybody have given up tool-based word processing, word processors still represent an important step forward for many users.

The COOP/Orm environment attempts to attack the problems outlined in the introduction with the following techniques:

- Representation of hierarchically structured documents.
- Integrated representation of user data and versioning information.
- Explicit version graph for browsing and comparing versions.
- Versioned bindings between documents.
- Support for parallel development and merge of alternatives.
- Active diffs for on-line awareness of changes by other users.
- Transparent distribution for users at different sites.

This paper is starting with a summary of requirements for an integrated version and configuration system for structured information. A description of the basic functionality and document model in our environment is presented in section 3. In section 4 we describe how configuration management can be introduced, based on this model. In section 5 we evaluate the functionality of our system and compare it with other attempts that go beyond traditional systems. In the following sections we summarize the status of our implementation, future work and our conclusions.

2 Terminology and requirements

We see software documents as highly structured information, preferably managed by an integrated environment. This view is in contrast to the common view of software as plain text files. The following list of requirements on support for versions and configurations has been significantly influenced by Katz work focusing on engineering databases for CAD/CAM systems [Kat90], which also demands support for highly structured information. Our view is similar to the one presented in [Kat90], but not identical, e.g. we see creation of versions as a more lightweight and frequent operation than he does. Katz also points out the need to limit the effects of a change to avoid a too large number of combinations of versions of documents, a problem we address explicitly below.

Terminology:

Information unit - smallest part of a document that is version controlled as one unit. This is typically a procedure (or even smaller: its interface, implementation, and documentation), or a paragraph of text in a structured text document.

Composite unit - hierarchically organized collection of Information units.

Document - semantically meaningful named Composite unit such as a module, a class, or a chapter in a book. A Document includes information about its version history, and all versions of the Information units.

Version - a snapshot of a Document. A Version can never be modified, but new Versions of the Document can be created. A Version has an *Originating Version* from which it is developed (the Originating version of the first Version of a Document is empty).

Variant - a special case of Version where there are several alternative versions developed from the same Originating version.

Delta - the difference between two successive Versions of a Document.

Change propagation - creation of a new version of an Information unit will trigger creation of new Versions of all Composite units including the changed Information unit, and in particular also of the Document it is part of.

Binding - relation between a Version of a Document and specific Versions of other (*Imported*) Documents.

Configuration - set of Versions of Documents related through Bindings.

Requirements on the storage model:

- Support for semantically meaningful named entities.
- Mechanisms to form Composite units out of more primitive parts.
- There should be support for configurations in the limited, tree structured case, since it is very common and offers important simplifications.
- It must be possible to include a Document in several different Configurations, thus the composition mechanism must support a general DAG (directed acyclic graph) structure rather than just Trees.
- From a Version of a Document it must be possible to determine the Version of all included Information units, local to the Document or Imported.
- A Document included in a Configuration through more than one path might as a result be included in more than one Version simultaneously.
- It must be possible to group changes (possibly to different Information units) into the same Delta. This requirement is motivated by the need to limit the number of created Versions and to represent logical changes.
- The model must support distributed development as transparent as possible.
- The model must support users' awareness of what other users are changing or have changed.

A configuration management system must also provide a good user interface and an efficient and compact implementation of the model.

3 The COOP/Orm hierarchical document model

We have chosen to support hierarchically structured documents directly since they are very frequently occurring and simple to represent and handle. Such documents can be seen as a kind of internal configurations where each unit as well as the configuration as a whole is version controlled together. In figure 1 we see an example of a typical structured document, a program with a class, its operations and documentation. The user interface with nested windows is described in [HM88]. The development history of the document is presented as a graph in a window. This graph can be used to browse the version history of the document, view particular versions of it and compare two versions, either sequential or further apart in time.

Updating a document involves three steps, (1) Selecting an originating version and creating a new version, (2) making a sequence of changes to one or several information units, and finally (3) terminating the update by 'freezing' the new version. Following the 'change propagation' scheme, all change to an information unit will go into the corresponding delta together with new versions of all composite units they are part of. Since a document is a tree structure, seen as repeated composite units, the change propagation ripples up to the top of the tree. As a result of this scheme, selection of a version of a document precisely determines the version of all information units of the document. The changes to a document can include changes to information units as well as to the structure of the document (adding/deleting units).

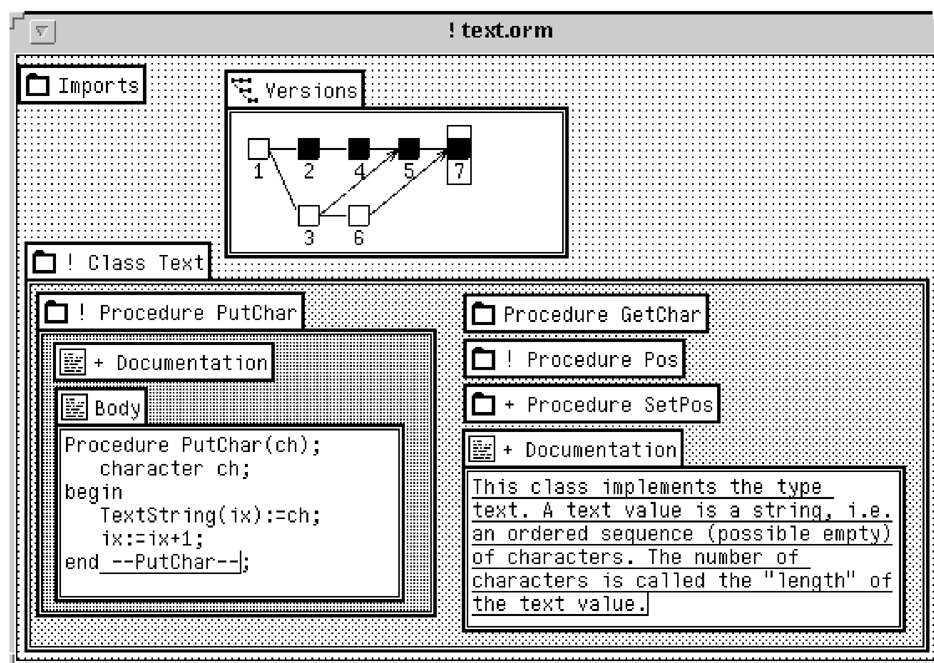

Figure 1 *Hierarchical document with version graph. We see version '7' of the document with marked differences compared with version '2' of the document.*

Version aware editor Hierarchical documents are browsed and edited with a specialized editor which allows the user to directly see the differences between versions of a document, both in terms of changes to an information unit as well as to the configuration itself, such as adding or deleting of units in the configuration. Creating a new version is seen as a comparison with its originating version and changes are highlighted as they are entered.

Explicit version graph An explicit version graph with a graphical user interface allows the user to view and browse the document in terms of its versions. In figure 1 we see the document in version '7'. In the same view we also show all differences relative to version 2. Signs like '+', '!' and '-' marks units that have been added, changed or deleted between the two compared versions. In open units we can see the detailed changes made, here added text is underlined and deleted text is overstroken. The editor is described in more detail in [Ols94]. The version graph also shows that the document has two alternatives, and changes from the alternative have been merged twice (indicated by the two arrows).

Local revision history All the information units in a hierarchical document share the same revision history, but a single unit might be unchanged (or even non-existent) in many of the versions of the document. This information is shown on demand in the 'local version graph' of the unit (an example for the 'Documentation' window of figure 1 is shown in figure 2). Here we can see that this particular unit did not exist in version 1,2,4 (dimmed boxes), thus created in version 3 and equal in some versions (3=5 and 6=7, marked with the double arrows). The user can thus browse the document both in terms of structure and versions at any level of detail at the same time.

Merge of variants Users working on the same document are free to create new versions and variants of the document. The editors offer strong support for merging of variants, suggesting default results and identifying conflicts for the user to solve [Ask94]. During merge changes to the contents of information units as well as changes to the structure can be handled.

Distribution A version of a document is never changed once it is established. The revision history of a document can only be extended with new versions. This means that it is not problematic to replicate a document in a distributed multi-server environment [MA95]. After a communication failure it is possible to synchronize the replicas. Merging of alternatives are done under user control, possibly later, in the same way whether the alternatives have been created in a distributed setting or not.

Awareness The version graph is shared by all users editing or viewing the same document. All creations of new versions of the document is thus immediately visible for all users (who have chosen not to close the version window). The granularity of presented

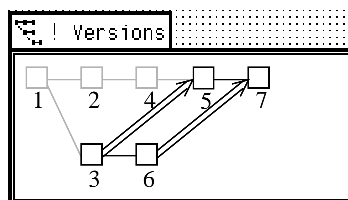

Figure 2 *Local version graph of a single information unit in a hierarchical document. The unit is not present in some versions (1,2,4) of the document, equal in some versions (3=5 and 6=7 respectively) and different when comparing others (e.g. 3 and 6, 5 and 7).*

changes is flexible, so a user can choose to be made aware of single changes as they are made. In [MM93] we have described how the model covers synchronization models from asynchronous to synchronous through the ‘Active diffs’ technique. This flexible awareness support can be provided also in a distributed situation.

3.1 Discussion

Hierarchical documents are intended to be used for representing relatively tightly dependent information. This might be programs, such as a class or a module with its operations, or a paper with sections and paragraphs. A document can include related information of different type, such as program code, documentation, users’ manuals, specifications, test cases, execution results and (as will be discussed further in the next section) bindings to other documents. The explicit representation of the version graph used for simple and fast interaction to view and compare versions helps the user to create a good understanding of the history of a document.

Integrated support for hierarchical representation offers a solution for the document size conflict. On one hand the Information units in a COOP/Orm document can be made relatively small in order to enable users to share small pieces of information and do work in parallel. On the other hand, the size of a COOP/Orm document can be chosen to be relatively large to collect logically related information in one unit.

COOP/Orm also offers a solution to the combination explosion problem since many related changes to the information units can be included in one version update of a document. If each information unit was represented as a single file with a version history of its own there would be a large number of combinations [Tic88], most of which would be inconsistent and uninteresting.

The versions of the document created in our model represent meaningful combinations of versions of the included information units, and the problem with many meaningless configurations is thus avoided. The scheme is not a restriction on the developer since it is always possible to create new variants of a document to include particular combinations of versions of the included information units.

The version control mechanism will register all versions of a document, also so called ’minor-revisions’, short lived versions during development of very little interest. In order to counter for a situation where the version graph grows beyond reasonable limits we are considering different mechanisms to collapse (and even remove) in particular long sequences of uninteresting versions of a document.

The underlying representation is using a backwards-delta technique and sharing of information for nodes that have not changed between versions [MAM93]. The use of character based, change oriented, deltas, rather than line based, might turn out to be more compact, in particular for the many small deltas we have to store. Our model also enables sharing of information between alternatives, which might be a significant improvement if there are many alternatives with small differences. There is no reason to believe that this representation form will be significantly larger than standard techniques used by SCCS and RCS, but it might turn out to be more compact.

The explicit shared version graph offers a powerful mechanism for awareness. It is possible to see in real time what other versions are created, and what the changes are. Together with the active diff mechanism it seems to offer mechanisms that covers the modes of interaction used in Software Engineering: mainly asynchronous, but in certain situations, such as initial design and debugging, also synchronous interaction.

The hierarchical document representation offers a mechanism for sharing documents between developers, but from a systems building point of view a document is one unit. In order to share a document between several systems we need also a binding mech-

anism to create configurations of documents. How to introduce such a mechanism with the document model outlined in this chapter is the main contribution of this paper, presented in the next chapter.

4 Configurations of hierarchical documents

A system can in principle be built as a single hierarchical document, but this would not allow use of a sub-component, such as a library, in several systems. In order to share components or sub-systems between systems, systems are built as a number of components that are then combined to make up complete systems. To provide mechanisms for this situation, a document can in our model contain bindings to other documents. Such a binding is targeted to a specific version of a document. We use the terminology that a document can *import* another document through a *binding*. As with a single hierarchical document, given a specific version of a configuration, it is always determined exactly which version of each imported document is included in the configuration. In figure 3 we can see the content of an 'Imports' window showing all other documents imported. For each document its name and version is shown. This view is called the *global version graph*, since it presents the versions of all documents the current document depends on.

A document can be included in a configuration through several bindings in the same or different configurations. A specific version of a configuration is always importing a specific version of the imported documents. A document that is included into a system through different paths might very well be included in more than one version at the same time.

Documents that contain no external bindings can be seen as a special case, constituting leaf nodes in the DAG of imported documents. Documents which are not further imported are called *systems*, while the intermediate case, documents with both incoming and outgoing bindings, are called *sub-systems*. Apart from the existence of bindings there is no difference between these kinds of documents.

Updating bindings

Changes to external bindings (as well as changes to other units) in a hierarchical document will propagate a version change to the top of the document, establishing a new ver-

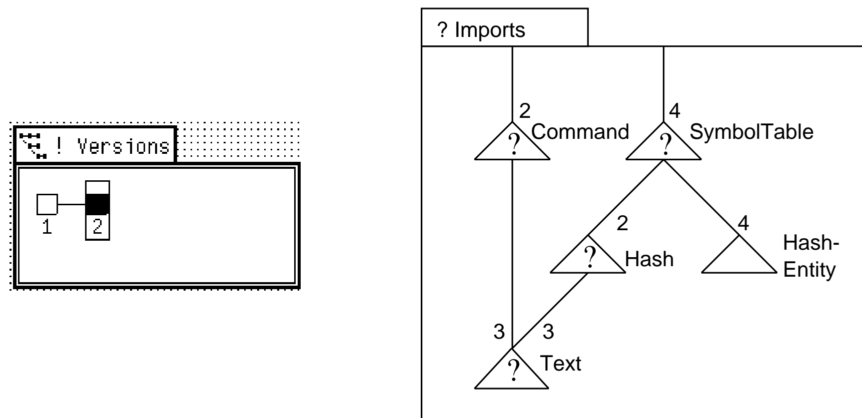

Figure 3 *Imported documents, directly and indirectly, in version '2' of a document. We can see that the document 'Text' is imported in two different ways. The marking, '?', indicates that there is a more modern version of the imported document available.*

sion of the document, as described earlier, but not penetrate through to the, possibly, large number of documents that *import* the document. Changing a binding to import a different, often newer, version of a document is something the developer frequently wants to be in control of. This operation is therefore an active choice (although supported by the user interface) rather than an automatic feature. Rebinding an import to another version of the imported document can be done by editing the external binding, or by editing in the global version graph. In both cases, updating an external binding always means creating a new version of the importing document. In figure 4 we show two steps in re-binding the version of an indirectly imported document. The 'open lid' in the version graph indicates a version under construction. We have decided to use the updated version, '4' of the document 'Text' and as a side-effect we have created new versions also of the intermediate documents. In the left hand situation the graph shows the difference in bindings between two versions of the configuration(2 and 3). The marker '?' marks information units where a newer version of the document is available and markers '!' indicate changed units. Version ranges (like '3-4') are used to show the versions bound to in changed units. Double drawn icons indicate that multiple versions of the document are considered in the current situation.

In the right hand picture of figure 4 the new version, 3, of the configuration is ready and the graph shows the resulting bindings of the document. Note that the developer, in this case, chose to use two different versions of the document 'Text' and the availability of the more modern version is still signaled by the '?' markers which is propagated upwards in the graph.

4.1 Discussion

The COOP/Orm document model extends well to supporting configurations of documents. The requirement to always be able to reconstruct versions of documents in a con-

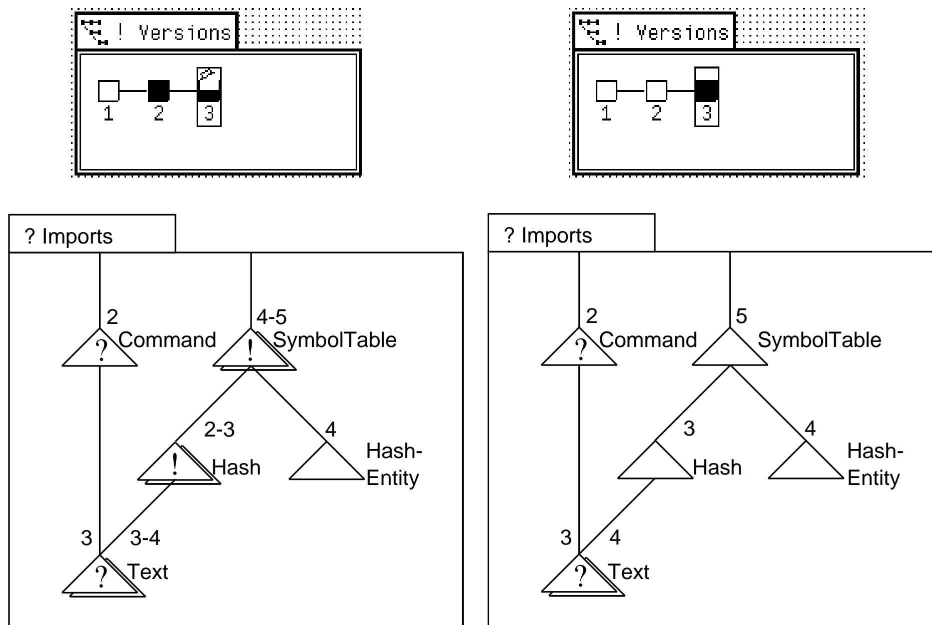

Figure 4 *A comparison of versions of a configuration being edited (to the left) and the final result, showing import of a document in multiple versions.*

figuration is fulfilled since bindings are to explicit versions of imported documents. In the terminology of [Tic88] our configurations are always 'baselines' and 'generic configurations' are ruled out by the requirement. Generic configurations are, despite the drawback not to fulfill the reconstruction requirement, motivated by the need to, in a flexible way, express selection of versions (like the last stable version) during development. There is also a need to select among the few meaningful combinations of the enormous number of possible configurations generated by combinatorics. The combinatorical explosion is created by the large number of individually versioned documents. Our approach is to try to avoid rather than solve this problem.

In our model information units are grouped together into documents with one and the same version history. There is thus typically fewer (but larger) documents when our model is used. Furthermore, since there is support for grouping of related changes, there will be much fewer versions of this document than the combination of versions of all included information units. These versions represent meaningful combinations of versions since they are created by the developer of the document. These two effects thus greatly reduce the number of combinations that need to be considered.

Attempting to further reduce the number of versions of a document that has to be considered by a developer who imports the document, one can envision techniques based on status of the versions, such as 'released', 'tested' etc. A filtering mechanism could then be used to show only a subset of the existing versions to the users importing the document. Seen from the developers viewpoint, this technique would solve the problem in a similar way as generic configurations, although the binding is done earlier.

A potential problem with our model is the 'snowball' effect that can be triggered when shifting to use a new version of a basic document, used (directly or indirectly) by many documents including the final system. Each of these needs to be extended with a new version in order to make the change take full effect. The binding update mechanism described above, has been designed to make it simple to create these new versions of a large number of documents. The filtering mechanism, outlined above, would help further in automating such regular updates.

We conclude that our model cover the needs addressed by both baselines and generic configurations. Baselines are trivially supported, while an automated re-binding mechanism covers the selection problem. The difference is that the selection is triggered explicitly in our model and as a consequence the requirement to be able to reconstruct systems is fulfilled.

5 Evaluation

The model presented in this paper will be evaluated first of all for the expressiveness and ease of use of the functionality it offers. The efficient implementation of the underlying representation of hierarchical documents has already been reported on in [MAM93]. The model meets the requirements listed above and thus also the requirements in [Kat90] as we interpret them in these circumstances. For further evaluation of the expressiveness we will use the list of questions relevant for software configuration management presented in [Tic88]:

Identification: *Identifying the individual components and configurations is a prerequisite for controlling their evolution. Questions: This program worked yesterday - What happened. I can't reproduce the error in this configuration. I fixed this problem long ago - Why did it reappear? The on-line documentation doesn't match the program. Do we have the latest version?*

Our model is storing all information in a versioned form and clearly supports the demand for identification. A system can always be recreated in any version. Our model also supports interactive presentation of differences between comparable versions and thus helps not only in identifying versions, but also differences both regarding content and structure of documents and configurations. The model supports integrated storage of code and documentation.

Change Tracking: *Change tracking keeps a record of what was done to which component for what reason, at what time and by whom. Questions: Has this problem been fixed? Which bug fixes went into this copy? This seems like an obvious change - Was it tried before? Who is responsible for this modification? Were these independent changes merged?*

Our model immediately answers the questions related to what was done to which components at what time and by whom. Remains to support questions on for what reason. Here we suggest a technique where bug reports are stored as individual documents and used to store bindings to related documents needing update. This is a technique similar to 'projects' in other systems, directly supported by our model. This technique gives a partial answer to the questions, the difficult one is - 'What bug fixes went into this copy'. It could be envisioned to be solved with a search mechanism over such Bug report documents and their external bindings.

Version Selection and Baselining: *Selecting the right versions of components and configurations for testing and baselining can be difficult. Machine support for version selection helps with composing consistent configurations. Questions: How do I configure a test system that contains my temporary fixes to the last baseline, and the released fixes of all other components? Given a list of fixes and enhancements, how do I configure a system that incorporates them? This enhancement won't be ready until the next release - How do I configure it out of the baseline? How exactly does this version differ from the baseline?*

Our model offers explicit choice of modules and versions to include in a configuration (=baseline). Creating versions of configurations as these examples are done interactively and under explicit control by the user. The system can also show differences between versions of configurations, to answer the last question.

Our model is designed to support tracing of the exact version of all components that go into any configuration. Selection of configurations is explicit and manual in our system, but chosen from a much smaller set of meaningful combinations than in a traditional situation. We have also shown how the model could support an automatic selection mechanism.

Software Manufacture: *Putting together a configuration requires numerous steps such as pre-and post-processing, compiling, linking, formatting and regression testing. SCM systems must automate that process and at the same time should be open for adding new processing programs. To reduce redundant work, they must manage a cache of recently generated components. Questions: I just fixed that - Was something not recompiled? How much recompilation will this change cost? Did we deliver an up-to-date binary version to the customer? I wonder whether we applied the processing steps in the right order. How exactly was this configuration produced? Were all regression tests performed on this version?*

These are issues that we regard as the responsibility of the software development environment. The environment we are working with, and where the model we have present-

ed is used, is highly integrated and spares the user the burden of controlling the "manufacturing process" itself. These tasks are done automatically and incrementally as needed.

Managing Simultaneous Update: *Simultaneous update of the same component by several programmers cannot always be prevented. The configuration management system must note such situations and supply tools for merging competing changes later. In so doing helps prevent problems like the following: Why did my change to this module disappear? What happened to my unfinished modules while I was out of town? How do I merge these changes into my version? Do our changes conflict?*

Our version control model is designed to support teams of programmers doing simultaneous updates. The model is that each member is free to create new versions and the system provides support for merging alternatives together using a operation-based diffing approach. Our model in fact goes beyond the requirements here in that it also supports a synchronous editing mode where programmers can see each others work in progress [MM93].

We thus conclude that the functionality needed to provide support for configuration management is covered in our model. We go beyond these requirements when it comes to support for teams of programmers.

5.1 Comparison with other systems

CVS [Wat, Ced93] is a system built on top of RCS and provides a central *repository* which contains *modules*, groups of files, of for instance program sources. The files are hierarchically structured (Unix files and directories). Every file is version controlled and branches as well as merge can be done on a single file.

The idea of module is matched by our hierarchical documents. CVS only uses modules as the configuration structure and not for version control while we use the structure for change propagation, compact storage and advanced presentation of differences. In CVS the revision numbers live a life of their own and it is optional *tags*, symbolic names attached to a certain revision of a file, that support versions of configurations. Thus, even if the structure of the repository is hierarchical the selection of files creating a configuration is not hierarchical. Our change propagation mechanism also enables diffs on configurations, which is not possible in CVS.

Furthermore, our approach supports a finer granularity of version control than CVS does. Our information units are intended for smaller pieces of information than what is usually stored in a single file. Also our model encourages creation of alternatives for development work, the result of which is then merged into the main development line (matched with the *commit* command in CVS). Minor revisions are thus saved in the developer's own alternative enabling a more detailed history log. In this way we support fine grained versions for each developer, without cluttering the main development line with too many versions.

Our integrated architecture also gives a more supportive user interface with graphical presentation of the version graph, and visibility of operation-based diffs (rather than whole source lines).

Teamware is a system designed for use in a distributed setting, supporting a workspace metaphor. Configurations, i.e. directories with files, can be manipulated and copied for independent development and later synchronization. Parallel development is thus supported in the copy-merge style. Merging of workspace copies is supported in that

conflicts are detected, i.e. updates that have taken place after the check-out, and have to be merged into the local workspace (and presumably tested) before an update can take place. If exactly the same file has been updated in both places, a three-way merge tool is used to support textual merge of the changes. The effect is that development of a workspace is a linear sequence. Alternative development lines are handled through use of multiple workspaces. Teamware is built on top of SCCS and a workspace includes copies of related SCCS files, which thus are copied and later merged. This enables movement of entire workspaces between file-systems and thus distribution, but with no awareness.

The COOP/Orm Multi-Server architecture enables replication and wide distribution, but also synchronization and awareness as long as there is network connection. Synchronization of SCCS files matches synchronization of replicated Orm documents after a network failure. The workspace model supported by Teamware matches Orm documents with one alternative for each Workspace. Seen this way Teamware insists on a particular merging order between alternatives that could be used also in COOP/Orm, but is not insisted on. In addition Teamware works with the latest version in each workspace, while in COOP/Orm the full development history is always available.

Continuus/CM is a process-driven, client-server system for change and configuration management. The Continuus/CM object management system is built on an inheritance-based type system, with several pre-defined object types. The *directory* and the *project* type represent grouping and configuration respectively, and can be mapped to the COOP/Orm document and configuration. There are, however, some differences. A new version of a *directory* object is only necessary if the set of objects changes (e.g. a new file is added to the directory). I.e., new versions of a file already in a directory can be created within the same directory version. This is in contrast to documents in COOP/Orm, where a change to a leaf unit propagates to be considered a change of the document. The drawback of Continuus solution, as we see it, is that the directory abstraction can not be used for selection of configurations. Instead of selecting a specific version of a directory, every object must be treated individually, which increases the complexity also when the configuration is later viewed. Both systems can compare versions of information units. However, the operation-based deltas used in COOP/Orm and the integration of editor and storage model gives a more fine-grained diff. Additionally, versions of both COOP/Orm documents and configurations can be compared and merged, highlighting the differences. This is, to our knowledge, not possible with directories and projects.

ClearCase is a version-driven configuration management system. It versions file-system objects, including files, directories, and links, of which directories are treated similarly to directories in Continuus/CM discussed above. Configuration control is supported through *views* which is a set of directories and files selected by a set of user-defined rules. Like COOP/Orm, ClearCase makes it easy to create a variant of a file and to reintegrate the work done into other lines of development. The merge is similar to ours, using the ‘common ancestor’ to find the changes between the common ancestor and each of the versions being merged. Both systems find the parts of the file changed in both branches and highlights them as conflicts. However, when ClearCase merges a file COOP/Orm merges a document, including merge of the document structure at the same time. Our merge is also more fine-grained because we use the operation-based deltas

given by the editor instead of comparing files, calculating the differences. This, we think, is the drawback of the transparency approach to integration taken in ClearCase. COOP/Orm also has more comprehensive group awareness, supporting the spectrum from shared version graph to active diffs enabling synchronous work.

CoVer [HH93,Web92] is a hypermedia version server implemented on top of the HyperBase hypermedia engine [SS90]. Versioned objects are represented by *multi-state objects* (mobs), which is a composite holding references to all states of the versioned object it represents. CoVer does not impose a fixed structure on the versions of a versioned object and version selection is based on viewing and browsing versions with respect to values on their attributes or relationships to other objects. General queries are used to find objects of a particular version. If a query returns several versions, they are considered as *alternatives* with respect to the equality characteristic given by the query.

The free format of version control in CoVer is in contrast to our approach where the evolution history is explicit. Versions represents the evolution and alternatives parallel development, i.e. a directed acyclic graph (DAG) structure.

In CoVer a *Task* maintains the (versions of) various objects used and created in the context of performing a job. A Task is implemented as a composite holding references to the objects determining its current state. This approach resembles the AND/OR graphs model presented in [Tic88], where a *mob* matches an OR-node and *Task* composites are AND-nodes. Our configuration mechanism can be used to form a collection for the purpose of performing a job, thus recording and preserving the information on which documents needed updates (and exactly which updates).

6 Implementation

6.1 Operations in the model

In this section we will summarize the functionality of the presented version model in form of the operations which the model supports. The representation of a hierarchical document has three components: revision history, user data, and external bindings.

Revision history
The revision history contains information to represent each revision of the document and their relations. The information is sufficient to create the version graph presented in the user interface. When a new version is created, this information is extended. The information format is a sequence of version pairs representing each arc in the graph.
Operations:

- Read version graph - return information describing the revision history of the document.
- Set focus - change currently viewed version of the document.
- Set compare - change version to which the viewed version is compared.
- Create version - create a new version succeeding from an arbitrary selected (but established) version. If the originating version already has succeeding versions a variant is automatically created. This is an operation that involves a long transaction, it is terminated explicitly and the version is ‘frozen’.
- Freeze version - make the version immutable.

- Merge versions - merge two versions creating a third.

User data
The user data constitutes the content of an information unit in one of its versions. The storage format uses backward operation oriented deltas and extensive sharing of common subparts [MAM93].
Operations:

- Read a unit - get the unit information in the version currently in focus. It returns application data for leaf units, and configuration information for branch units.
- Read a unit delta - get the delta information for a unit representing the changes between the focus and compare versions.
- Write a unit - update the unit information of a unit in the version in focus, and at the same time update the delta to its originating version(s).
- Add a unit - the document configuration is expanded with one new unit. Its type can be a composite unit or leaf unit (e.g. text or external binding).
- Delete a unit - delete a unit in the configuration.

The operations, Write, Add and Delete a unit can only be called when a new version is in creation (between the Create and Freeze operations). Add and Delete a unit are actually edit operations on the containing composite unit. The actual content of an information unit depends on the type of the unit. Here we have mentioned the fundamental node types composite and external bindings (as of below), and nodes of type text. The model can, however, handle user data of any type.

External bindings
The external bindings contain the path and version of all external documents this document depends on, or imports. This information is stored and version controlled as part of user data, but considered a special component due to its importance for configuration management.
Operations:

- Read external bindings - return bindings to external document (including version).
- Read binding deltas - return changes to the bindings between the focus and compare versions.
- Create external binding - add an import of a specific version of a document.
- Delete external binding - remove an import of a document.

The operations Create and Delete uses the operation Write unit and can thus only be called during creation of a new version.

6.2 Implementation status

The implementation of the model is carried out as part of a software development system with the ambition to support teams of users working together on shared information. The implementation is organized as a multiple server multiple client architecture. In the first version it supports a semi-structured representation of programs and text. The hierarchical document supports structuring a document as a tree of information units, but each unit

can currently only contain unstructured text. A text editor with version control mechanisms provides the means to browse and navigate versioned text.

6.3 Future work

Currently the version graph interface shows all existing versions of a document. This clearly does not scale up to handling systems with a long and complex history. We are therefore working with methods to suppress some of the information in the graph (hiding details like long sequences and older parts) although the user can always 'open' these parts to see all the details as he wish. Similarly, the global version graph shows all included document in a configuration. This does not scale up to large systems with many components. Here we are considering to suppress deeply nested graphs (again allowing the user to explore these parts if he wishes), but still reporting the status of these parts (propagating the '?' markers). Operations are also needed to perform related rebinding operations over a collection of modules in one operation (like switching to use a new release of a popular module without touching all the importing modules one by one).

In future versions we will also provide a versioned editor for abstract syntax trees for integration of the fine-grained version mechanism in the Orm programming environment.

The protocol for server-server communication has been designed but is not yet implemented.

Finally we see the system as an interesting environment for experimenting with process support. The explicit version mechanism is initially motivated for enabling awareness in a cooperative editing environment and a process support system is by nature a CSCW system. We also expect that interactive language development support mechanisms developed in the Mjølner/Orm environment (and available in COOP/Orm when the abstract syntax editor is available) will be useful to develop support for specifying processes.

7 Conclusions

We have presented an integrated model for fine-grained version control and configuration management. The design is exploring a two level approach. A restricted grouping mechanism is supported inside tree structured, hierarchical documents, offering automatic version propagation. Between documents a general binding mechanism is offered with explicit control over version selection of imported documents.

Supporting hierarchical documents have several benefits. At the same time it offers a configuration mechanism for keeping related information together in a tree structure, a fine-grained revision control mechanism, and automatic change propagation. A direct manipulation user interface allows the user to browse a document both according to its structure and its versions presented in a graph. Differences between versions of a document can be interactively constructed and presented both regarding changes to structure and contents.

Through external bindings hierarchical documents can be related in DAG structures, a document can thus be used in many places. In a single hierarchical document, change propagation is automatically generating new versions of containing configurations in the tree structure. Between documents the change of bindings to different versions is explicitly managed by the user, but guided with a graphical user interface. Bindings are always targeted to a particular version of an external document and the bindings are stored as part of the versioned information of a document. As a result it is always possible to recreate all the documents included in a configuration in their correct versions.

The model is initially intended for supporting development in integrated environments, but is also useful in other settings. The model has been developed to support groups of developers sharing information and is offering support for both synchronous and asynchronous modes of editing, parallel development and strong support for merging of variants. The details of these aspects are covered in other papers [MM93, MAM93, Ask94, Mag95].

Acknowledgments

The authors wants to thank all the members of the software development research group at Dept. of Computer Science, Lund Institute of Technology, for stimulating discussions which have contributed substantially to the work presented in this paper. In particular, we want to thank Torsten Olsson who is working on the structured document editor and Görel Hedin for constructive comments on earlier drafts of this paper.

The work presented in this paper was supported in part by NUTEK, the Swedish National Board for Industrial and Technical Development.

References

[Ask94] Ulf Asklund. Identifying Conflicts During Structural Merge. In Magnusson et al. MHM94.

[Cla95] Dave St Clair: Continuus/CM vs. ClearCase, URL: http://sunsite.icm.edu.pl/sunworldonline/swol-07-1995/swol-07-cm.html, SunWorld Online, 1995.

[Ced93] Per Cederqvist. Version Management with CVS. Available from info@signum.se, 1993.

[Gus90] A. Gustavsson. *Software Configuration Management in an Integrated Environment*. Licentiate thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 1990.

[HH93] Anja Haake and Jörg M. Haake. Take CoVer: Exploiting Version Support in Cooperative Systems. In *Proceedings of INTERCHI'93*, ACM Press, Amsterdam, The Netherlands, April 24-29 1993. Addison Wesley.

[HM88] G. Hedin and B. Magnusson. The Mjølner environment: Direct interaction with abstractions. In S. Gjessing and K. Nygaard, editors, *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 41–54, Oslo, August 1988. Springer-Verlag.

[Kat90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4), December 1990.

[KLMM93] J.L. Knudsen, M. Löfgren, O.L. Madsen, and B. Magnusson, editors. *Object-Oriented Environments - The Mjølner Approach*. Prentice-Hall, 1993.

[LvO92] Ernst Lippe and Norbert van Oosterom. Operation-based Merging. In H. Weber, editor, *SIGSOFT'92 Proceedings*, Tyson's Corner, Va., December 1992. ACM. SIGSOFT Software Engineering Notes, 17(5).

[MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December 1993.

[MHM94] Boris Magnusson, Görel Hedin, and Sten Minör, editors. *Proceedings of the Nordic Workshop on Programming Environment Research*, Lund University of Technology. LU-CS-TR:94-127, Lund, January 1-3 1994.

[MM93] Sten Minör and Boris Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work*, Milano, Italy, 1993. Kluwer Academic Publishers.

[MA95] Boris Magnusson and Ulf Asklund: Collaborative Editing - Distributed and replication of shared versioned objects. Presented at the Workshop on Mobility and Replication, held with ECOOP 95, Aarhus, August 1995. Available as: LU-CS-TR:96-162, Dept. of Computer Science, Lund, Sweden.

[Mag95] Boris Magnusson: Fine-Grained Version Control in COOP/Orm, Presented at the Workshop on Version Control in CSCW, held with ECSCW'95, Stockholm, Sept. 1995. Available as: LU-CS-TR:96-163, Dept. of Computer Science, Lund, Sweden.

[MHM+90] Boris Magnusson, Görel Hedin, Sten Minör, et al. An Overview of the Mjølner Orm Environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, Paris, June 1990. Angkor.

[Ols94] Torsten Olsson. Group Awareness Using Fine-Grained Revision Control. In Magnusson et al. MHM94.

[RBI95] W. Rigg, C. Burrows and P. Ingram: Ovum Evaluates: Configuration Management Tools, Ovum Limited, London, 1995.

[Roe75] M. J. Roekind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.

[SCC] *SCCS - Source Code Control System. UNIX System V programmer's Guide*. Prentice-Hall Inc. pp 59-700.

[SS90] Helge Schütt and N. Streitz. HyperBase: A Hypermedia Engine Based on a Relational Database Management System. In A. Rizk, N. Streitz, and J. André, editors, *Proceedings of the European Conference on Hypertext (ECHT'90): Hypertext: Concepts, Systems, and Applications*, Cambridge Series on Electronic Publishing, pages 95–108, Versailles, France, November 27-30 1990.

[Team] TeamWare user's guides, Sun Microsystem, 1994.

[Tic85] Walter F. Tichy. RCS - a system for revision control. *Software Practice and Experience*, 15(7):634–637, July 1985.

[Tic88] Walter F. Tichy. Tools for software configuration management. In *Proceedings from International Workshop on Software Version and Configuration Control*, Grassau, Germany, February 1988.

[TJ88] Dave Thomas and Kent Johnson. Orwell: A Configuration Management System For Team Programming. In N. Meyrowitz, editor, *Proceedings of OOPSLA'88*, San Diego, Ca., September 25-30 1988. ACM. SIGPLAN Notices, 23(11).

[Wat] Gray Watson. CVS Tutorial. Available from gray.watson@antaire.com.

[Web92] Anja Weber. CoVer: A Contextual Version Server for Hypertext Applications. In *Proceedings of ECHT'92*, November 30 - December 4 1992.