The Unified Extensional Versioning Model

Ulf Asklund[1], Lars Bendix[2],
Henrik B. Christensen[3], and Boris Magnusson[1]

[1] Department of Computer Science, Lund University, Sweden. {ulf | boris}@cs.lth.se
[2] Department of Computer Science, Aalborg University, Denmark. bendix@cs.auc.dk
[3] Department of Computer Science, University of Aarhus, Denmark. hbc@daimi.au.dk

Abstract. Versioning of components in a system is a well-researched field where various adequate techniques have already been established. In this paper, we look at how versioning can be extended to cover also the structural aspects of a system. There exist two basic techniques for versioning - intentional and extensional - and we propose a unified extensional versioning model for versioning of both components and structure in the same way. The unified model is described in detail and three different policies that can be implemented on top of the general model are exemplified/illustrated by three prototype tools constructed by the authors. The model is analysed with respect to the number of versions and configurations it generates and has to manage. Finally, the unified extensional model is compared to more traditional intentional models on some important parameters. The conclusions are that the unified model is indeed viable. It not only provides the functionality offered by the intentional model with respect to flexibility during development and management of combinatoric complexity, but also offers a framework for management of configurations that enables systems to provide much more advanced support than is commonly available.

1 Introduction

Many models for configuration management [Tic88, CW98], as well as available tools, e.g. ClearCase [Clear] and CVS [Ced93], make a clear separation between how they handle atomic entities (versioned objects, modules, etc.) and composites (configurations, libraries, systems). In these models, atomic entities are version controlled individually while configurations are formed by applying *selection mechanisms*. When considering all atomic entities at the same time the number of possible combinations of their versions and variants is overwhelming. Using rules (such as the 'latest') is an attempt to automate this selection process.

We present experience from using another model, the *Unified Extensional Versioning Model,* where both atomic entities and configurations are version controlled extensionally. This model relies on a mechanism, *version concentration*, to reduce the number of combinations that need to be considered. These combinations, i.e. versions of configurations, that arise are the subset of versions that the user actually explore and are thus the ones that are, or have been, interesting and meaningful for the user.

The unified extensional versioning model has several interesting features compared to models that combine extensional versioning of atomic entities and intentional selection of configurations:

- Simple conceptual framework: Basically developers/users have to learn only one single concept namely "a version of an entity".
- Modularization principle: As any version of an entity embodies the bound sub-configuration that is rooted in it, it allows the well-known principle of encapsulation and abstraction to be applied at the SCM level.
- Scaleability: It is general in the sense that it can be used recursively - configurations can form parts of new configurations.
- Consistency: The versions of configurations that are encountered are the ones explicitly created, and thus likely to be interesting; in contrast to a combination generated by a (potentially flawed) ruleset.
- Architectural traceability: A configuration in a particular version uniquely defines the version of all its constituent parts; thereby not only the evolution of atomic data (like source-code or document text) is traced, but just as important also the evolution of the very structure and architecture of the system, that is, changed, added, and deleted relations between entities.

Elements of the unified extensional versioning model has independently been developed and explored by the authors while building three different systems with different aims and motivations.

COOP/Orm is a configuration management prototype system that has been designed explicitly to support development in distributed groups, people that work tightly together, but are geographically distributed [Ask99, AMP99]. Some of the demands from this situation - collaborative awareness and supporting both synchronous and asynchronous interaction - have been reflected in the design: a very explicit version control system, support for fine-grained version control and support for hierarchical structures.

CoEd is a research prototype for supporting collaborative writing of hierarchically structured documents. It was designed to solve the specific problems students had when writing their semester reports. In particular, focus is put on providing overview of the document and communicating information through version histories. Furthermore, it is possible to directly modify the structure of the document under full version control.

Ragnarok is an architecture based software development environment prototype. The logical architecture of a software project is used as framework for version- and configuration management; this leads to an SCM model that minimizes the gap between the architecture oriented design domain and the SCM domain. Furthermore, it provides strong traceability of the architectural evolution.

In all three systems we have arrived at models that share a common foundation. In some cases, the prototypes we build do not support all the same functionality. This is because we have focused on different aspects but in each case the systems can (at least in principle) be extended to support the full unified extensional versioning model. We thus argue, that the model has proved useful in these different situations and has a value also in its generality, and simplicity.

In the rest of this paper, we first characterize the traditional models and identify some of their drawbacks. Then we present a *unified extensional versioning model*, both in its general version and the particulars of its three different instantiations. In chapter 4, we discuss the implications of the model and compare it with the intentional model. Finally, we draw our conclusions.

2 Existing Versioning and Configuration Models

One of the fundamental problems when dealing with configurations is that with already a small number of components - each in a number of versions and variants - the number of possible combinations get very large. Mathematically, the number of combinations grow exponentially with the number of components and versions and any attempt to deal manually with **all** of them is unmanageable. This problem of combinatorical explosion has to be dealt with in every model. First we survey and evaluate existing solutions before presenting our model in the next chapter.

At this point we also need to be precise about how we use the term 'configuration'. *A configuration is a named collection of atomic entities and other configurations*. Two versions of the same configuration may differ in that they include different entities and/or the same entity in different versions. Other authors would see what we call ‘versions of configurations’ as different configurations. We see a set of selection rules as a *specification of a configuration* while others would identify the specification with the configuration it might result in. In our terminology a configuration is always bound, while a configuration specification can be bound or generic. Our use of the terms is consistent with the common CM-view on atomic entities where a file has identity and might exist in several versions (which is in contrast to non-version-aware tools that see different versions as different files).

2.1 Dealing with configurations - Intentional versioning

Many existing CM-systems (ClearCase, CVS, etc.) and models use what is sometimes called 'intentional versioning' of configurations in order to handle the problem of combinatorical explosion. The approach builds on formulating selection rules which are then used to choose the particular variant and version of an atomic entity. Often these rules are evaluated on demand when the atomic entity, a file, is needed - for viewing, editing or translation. Although this approach is one way to limit the selection problem, it has some drawbacks.

- The representation of a configuration is indirect, embedded in the formulation of the rules (e.g. in a small script file), and in the build information (often in a 'makefile'). Given such a rule-based specification, the only way to find out what the configuration really is, in terms of what files are included and in what versions, is to actually build it and register the result.
- Differences between configurations in terms of what files are included in what versions are hard to find out since that can not be deduced from comparing the sets of rules. The only way to find out is to evaluate the different sets, register them and then compare the results.

- Consistency is hard to guarantee since incompleteness or 'errors' in the rules may go unnoticed for a long time, and only show when a new version of some file is created and then result in an unintentional (wrong) configuration. As a consequence there is never a guarantee that a given rule will result in the same set of files in the same versions when evaluated at a later time. For important configurations, such as releases, it is often paramount to be sure that all included files can be found and recreated in exactly the relevant version. As a safeguard all files included in such configurations are often copied and stored separately.
- Tagging is a way to label versions of individual files and when used methodically can be used to pin the files and their versions as included in a configuration. Unfortunately this is a rather primitive mechanism since there is not always a guarantee that such lables are not changed afterwards. There is no support for relating configurations registered in this way to each other or to calculate the difference between them.
- The rules can include generic facilities such as selecting the 'Latest' version of a file which change over time, resulting in so called 'generic' configuration specifications. The same rule-based specification of a configuration can thus over time result in many different resulting configurations. This mechanism can thus be seen as a further way to limit the effects of the combinatorical explosion problem, but it creates a new problem since it defeats traceability. It is impossible to guarantee that the same system will be build from the same generic rules at a later time. In the extreme case one can not be sure that the versions of the files just compiled are the same as the ones viewed in an editor.

Intentional versioning of configurations is used by most traditional, state-based CM systems as well as by change-based systems. Based on the analysis above and the obvious lack of support for rather common situations we have found it motivated to explore other ways to solve the problem of combinatorical explosion. Before outlining the solution presented in this paper we will first review how versioning of atomic entities, files, are handled.

2.2 Dealing with atomic entities - Extensional versus Intentional versioning

Change-based systems, e.g. Aide-de-Camp [Crn92, AdC90], COV [GKY91, MLG$^+$93], name deltas between versions of atomic entities rather than the versions themselves. An advantage of this mechanism is that the deltas can be combined in many more ways than there are typically versions in a state-based systems and also in ways not foreseen by the creators of the deltas. The possible combinations are somewhat limited by restrictions among some of the deltas that might exclude or presume each other, but the difference is still big. For example all versions that in a state-based system can be created through a trivial merge can here be created directly. A needed version of an atomic entity (a file) is put together on demand when needed (for viewing, editing, translating). In existing changed based systems this task is handled through rules and selection, thus using intentional versioning also for atomic entities.

In a change-based system the number of potential versions of each atomic entity is larger. The number of possible combinations is thus also larger. The combinatorical explosion problem of configurations thus gets even worse in change-based systems. In existing systems this problem is again handled through use of selection rules. 'Intentional versioning' is thus used consistently for atomic entities as well as for configurations.The criticism we formulated above for handling configurations with intentional versioning thus applies both when dealing with configurations and atomic entities. The change based approach might have other advantages, but when combined with intentional versioning as commonly done it does not improve on the situation described above.

In contrast to change-based systems, *state-based systems* use 'extensional versioning' when dealing with atomic entities, e.g. files. Extensional versioning means that all the versions of the entity are explicitly represented. They can for example be presented as a version graph and a given version can be retrieved by identity at a later time in exactly the form it was created. Versions of entities can be compared and related to each other, e.g. by the partial relation 'derived from'.The problems we listed above for configurations when using intentional versioning are thus not present when dealing with atomic entities using extensional versioning.

A fundamental criticism of traditional state-based systems is that they offer very different mechanisms for dealing with atomic entities and with configurations. Unfortunately this leads not only to proliferation of concepts, but also to a weak support for managing configurations.

New approach: The Unified Extensional Versioning Model
Traditional state-based systems and change-based systems are similar in that they use intentional versioning for handling configurations. In this paper we put forward a radically different approach - using explicit versioning also for configurations. We will show how we with this approach counter the problem of combinatorical explosion both in general, and further with different mechanisms in the three prototype systems we have built. The model we present also avoids the problems discussed above in connection with intentional versioning of configurations. Finally it has the advantage of offering one unified version model for atomic entities as well as for configurations.

	Atomic entities (files)	*Configurations*
Intentional versioning (rules)	Change-based systems	Change-based systems Traditional, state-based systems
Extensional versioning (explicit versions)	Traditional, state-based systems UNIFIED MODEL	UNIFIED MODEL

3 The Unified Extensional Versioning Model

In this chapter we first present the unified model, both from a somewhat formal perspective and illustrate with examples. We then describe how the model has been used in three systems we have built.

3.1 The model

Document model

A 'document' in this model is structured and the structure can be expressed in a grammar as shown in Figure 1. Relations between documents is also part of the model through the notion of links. 'Document' is here used in a general sense of a file, dataset, that can contain any form of information, e.g. program source, English text, graphics, etc.

D ::= T	*D - document (abstract node, non-terminal*
T ::= C|L|N	*T - tree (abstract node, non-terminal)*
C ::= T* [‘local data’]	*C - composite node (concrete node, production*
L ::= ‘name’ ‘version’	*L - link node (concrete node, production)*
N ::= ‘local data’	*N - atomic node (concrete node, production)*

Fig. 1. Grammar specifying the document structure

- N-nodes support storing data. It can be text, source code, graphics or any other information which is thus of no concern to the model. Different N-nodes can contain different types of data, so the model supports documents with mixed data.
- C-nodes support Composition, whole-part relations. This is introduced in recognition of the need for support of hierarchies commonly used to structure text documents (chapters, sections, paragraphs), programs (modules, classes routines) and many other kinds of information.
- L-nodes support Reference semantics, arbitrary relations between documents. This is introduced in recognition of a need to share common parts between configurations (libraries, modules, classes in programs, and illustrations, appendix, quotations etc. in textual documents). The 'name' attribute stored in an L-node is the information needed to link to another document. The 'version' attribute is the information needed to denote a specific version of the document which will be explained in the next section.

The model supports structure in two ways, through C-nodes and L-nodes. There is thus some redundancy in the model since composition, tree-structures, can be built out of a restricted use of L-nodes. The motivation to include C-nodes and explicit support for composition in the model is that tree-structures is a fairly common case and that we view composition and reference semantics as distinct cases.

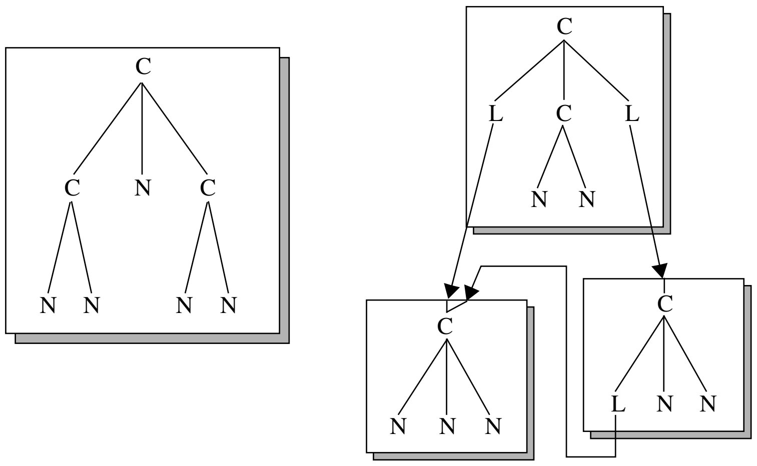


Fig. 2. Composite document and configuration represented in the Unified Extensional Versioning Model.

Traditional document models can be understood in our model as documents which only contain one N-node. Such models does not support internally structured documents and do not support relations between documents.

Examples of structured documents

Figure 2 depicts examples of document structures. The left hand example shows a single tree-structured document. The right hand example shows three structured documents linked together. Lines indicate composition in a document while arrows are references between documents.

A more concrete example of a tree structure is a book. The left hand example in Figure 3 depicts such a book consisting of three chapters, where chapter one and three both have two sections respectively. The relation between the book, the chapters, and the sections are 'consists of' or 'contains' and the total structure represents one entity - the book.

A concrete example of a structure also using L nodes is Java source code for an application consisting of classes and packages. The small right hand application in Figure 3 consists of one class and it imports two classes, A and B. The class-to-operation relation is of the same type as the relations used in the book, i.e. 'consists of' or 'contains'. The relations import-to-class and su.cl-to-class (super class) is, however, references i.e. links. It would e.g. be wrong to say that class B consists of class A. Moreover, both class A and B might be included in many other applications. The semantic difference between composition and reference semantics will also show up in versioning of documents discussed below.

Versioning

Both structure and contents of a Document will evolve over time. In the extensional model all node types (N, L, or C in the grammar) are explicitly versioned. Creation of a new version of a node is triggered by any of the following conditions:

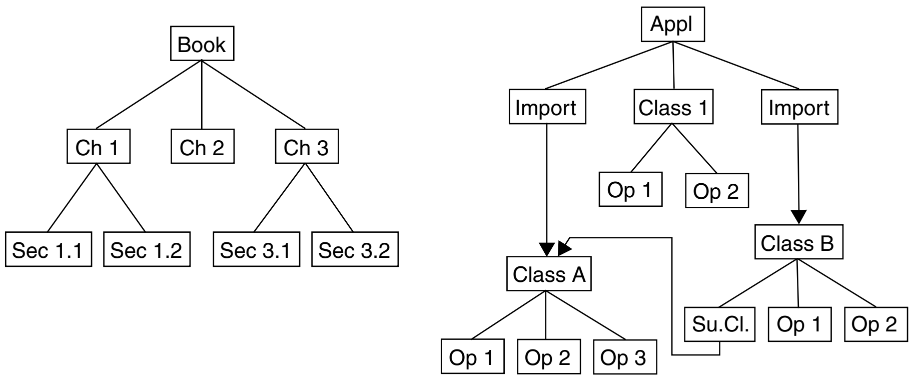


Fig. 3. Example of structured documents: A book and Java source code.

- N,C-nodes - a new version is created when its 'local data' is changed
- L-nodes - when name, or version is changed
- C-nodes - also when any of its sons is added, deleted, or changed

Changes to a document occurs during a 'session', a long transaction. The extent of a session is defined by the user who explicitly or implicitly controls when a session starts and ends. During one session there is created at most one new version of each node if needed according to the rules above. Repeated edits to local data in one node are thus part of the same change to that node. Several additions, deletions and changes to the sons of a C-node also result in only one new version of the node. The length of a session, and thus the amount of changes that go into the same version, can be used to control the granularity of the versioning. When a session is ended the created versions of the nodes can no longer be modified.

Versions are related through the derived-from relation and can form arbitrary DAG structures. The version mechanism thus can represent concurrent development and merge of Documents, atomic entities as well as configurations.

For a document a session means that a new version of the document is created. For each node changed during the session a new version of the node is created (but only one). The rule that C-nodes are considered changed also when only their sons are changed results in an effect know as 'change propagation' [Kat90]. Any change will result in new versions of all father nodes of the changed node up to the top node (if not already changed in the same session). The effect that there is only one new version of a father-node during a session can be seen as a *version concentration* mechanism.

This automatic change propagation mechanism for documents is consistent with how changes of compositions are perceived. For example a change to a paragraph in this paper means the whole paper is changed. It also means that a version of a document uniquely determines which internal nodes to include and for these which version.

For relations between documents the version attribute of an L-node determines the version of the referenced document. If another version of the referenced document is wanted the version attribute of the L-node needs to be changed (and thus the L-node itself, all enclosing C-nodes, and ultimately the document where it resides).

The model thus implies that updating a link to another (for example newer) version of a document means that the referencing document must be changed. When and how this is done is not specified in the model, but can be supported in a tool by different convenient mechanisms to administer updates between documents. Examples of such mechanisms are illustrated by the tools present below. Again the session mechanisms and long transactions can be used by the user to limit the number of such versions that actually occurs.

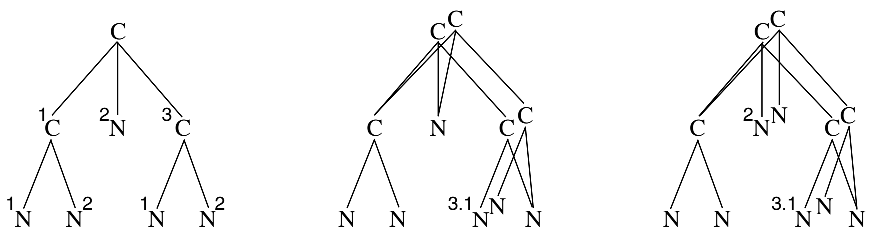
Example, versions of structured document

Figure 4 depicts the evolution of a tree structured document. In Figure 4b the local data in the N-node '3.1' (sons numbered from left to right) is modified and a new version of that node is created. As a consequence also a new (intermediate) version of its father node is created (node '3') and of the root node, i.e. the entire document is considered changed. In Figure 4c the user has continued the session by also modifying node '2', thus creating a new version of it. Since a new version of its father node already exists change propagation has no effect in this case. It is thus possible to make many related modifications to the document, all included in one and the same version of the document. The user controls when a session is ended and thus when and what versions are actually created.

An example where the structure shown in Figure 4 might arise is a book with three chapters, see Figure 3. A change in one of its paragraphs results in a new version of the book and so does several modifications during the same session. This situation is consistent with the situation that would arise if the versioning model would not acknowledge structure and the three chapters would be maintained as one single file. Versioning of compositions using change propagation coincide with the situation when more primitive composition mechanisms are used. A document can also be seen as a bound configuration of its nodes. Given a version of the document - the version of all its nodes are directly determined.

Example, versions of configurations of documents

In this example we consider a situation with three documents, one (D1) importing the other two (D2, D3) as shown in Figure 5. Modifications to D2 and D3 results in new versions of these, one for each session depending on how the user chooses to organize his work. In Figure 5 we show the situation after one edit session with D2 and two ses-



a) Initial situation of structured document
b) Node 3.1 has been changed
c) Also node 2 is changed

Fig. 4. Many changes within the same version.

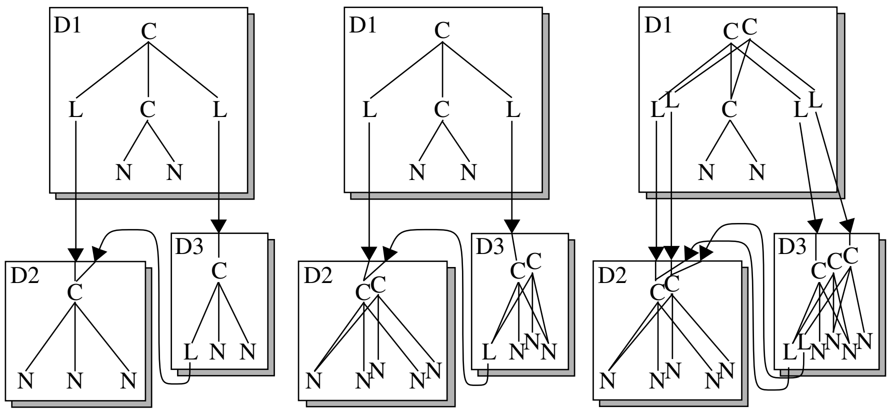


Fig. 5. Editing an L-node often means rebinding to a new version.

sions with D3. In order to use the newer versions of D2 and D3 also a new version of D1 needs to be created where its link nodes are changed. The user can here decide to move to the latest version of D2 and D3 (as shown in the Figure) or to use any other combinations of versions of D2 and D3. The structure is in this case a small graph, but links can be used to build higher trees and indeed arbitrary directed acyclic graphs and the same mechanisms applies. Situations where structures as the one presented in Figure 5 can occur is for example in software development where the documents are source modules, depending on each other such as in the situation illustrated in Figure 3.

Summary
We have presented the unified extensional versioning model and explained how it handles structured information, versioned relations between documents, and how the extensional versioning work for these documents. We have also shown how the combinatorical explosion problem is countered, by the use of long transactions, called sessions, and the effect that hierarchies limits the number of combinations of its components, version concentration.

It still remains to show how this model can be used in tools and to see if it is viable in practice. One can immediately foresee two potential problems. If, for example, sessions in practice are very short the number of versions created can still be very large and this would defeat the version concentration mechanism. Our experience presented below shows that this is not the case. Another possible problem would occur if the overhead to update a link to use a new version of a referenced document would be large, since this will be a fairly common operation. Again the presentation of our tools below show that this is not the case. Mechanisms to swiftly perform such updates over sets of files have been designed and tried out in practice.

3.2 Prototype implementations

In this section we will briefly present the three prototype system we have developed based on the extensional versioning model. We will concentrate on the aspects of the prototypes that are directly related to the model and ignore much of other, although important, aspects of the systems. The systems are not in all cases supporting the full model, but only need some of its facilities. They also differ in their interpretation of when and how versions are created. The three presentations illustrate how the model can be implemented and combined with different facilities to create flexible and easy to use systems for different purposes.

COOP/Orm

Background The starting point for the development of the COOP/Orm research prototype [MA93, MAM93, MA95, MA96] 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). From the CSCW perspective the requirements collaborative awareness and support for concurrent work have resulted in fine grained versioning (both spatial and temporal), optimistic check-out and strong support for merge. COOP/Orm is built as an on-line system with a built in editor. Everything is thus stored in the repository rather than in separate workspaces. SE issues have lead to better support of hierarchical structures, and sharing of common parts between applications.

Document model The COOP/Orm environment implements the grammar as defined in Figure 1. The size of a document is user defined. Typically the granularity of a document could be an article, a class, a package, etc. Within the document the granularity of atomic N-nodes can for example be a section, a method, or even the body of a method (e.g. dividing a method into head, body and documentation), as desired by the user.

Versioning of one document All changes to a document are made during a session, which involves three steps, (1) selecting an originating version of the document and creating a new version from it, (2) making a sequence of edits to, changing/adding/deleting, one or several nodes within the document, and finally (3) terminating the session by 'freezing' the new version. Both the creation and 'freezing' of versions are explicit operations by the user who thus determines the length of a session.

Versioning of configurations of documents Relations between documents are implemented through link-nodes, which are nodes within the internal structure of a document and can thus only be changed during a session. In a structure of documents each document has its own sessions, during which its link-nodes can be modified.

A user can choose to modify documents in short sessions thus giving detailed control and traceability, but new versions of configurations for each edit. It is also possible to use long sessions and let versions of documents remain open allowing many changes of their (link) nodes. If this model is mixed with the manual (more frequently freezing versions) a balance of strong version concentration and traceability can be obtained.

CoEd

CoEd [BLNP97, BLNP98] is a prototype environment that supports collaborative writing through the use of advanced version control policies. CoEd manages hierarchically structured textual documents only, where the relation between the parts is that of composition. This means that CoEd does not support the L-nodes of the general model. In the specifying grammar the L-production is removed and the T-production simplified accordingly: T::=C|N. When changes have to be propagated, new versions are created of all nodes on the path from the node that was changed to the root of the document.

CoEd works as a repository only, which means that the user cannot directly edit the bound configurations of the document, as they are immutable. So a traditional checkout-edit-checkin way of working has to be followed. A session starts when a structure is checked out from CoEd. It is possible to check out just a part of the document by indicating the C-node that forms the root of the subpart. When the (sub)structure has been checked out, a single file containing all the LaTex text for the (sub)structure will exist in the users file system. This file is mutable and the user can edit it as he wishes, changing even the structure of the document. After the editing, the file representing the (sub)structure is checked back into CoEd. The file is parsed and if it represents a valid LaTex structure, CoEd discovers what has changed. Changes are propagated all the way up to the root of the document. When the document is in the user's file system, its structure is not explicit anymore, but only indicated by the respective LaTex commands. However, whenever the document is inside the repository, its structure is explicit and it is kept as a series of versions of bound configurations that can be browsed and retrieved.

Even though CoEd has no explicit notion of a workspace, it does implement the possibility to work directly on the structure of a document inside the repository. If we want to 'promote' section 3.2 of this paper to become chapter 4, this can easily be done by dragging the section to the new chapter's place. This creates a new bound configuration of the document, where section 3.2 is deleted from its original place in the structure and inserted at the new place. Presently, there is no explicit session concept when working inside CoEd's repository even though all changes are versioned. This means that if we make several modifications to the structure this will result in several new bound configurations being created, even if they might conceptually be considered as one change.

Ragnarok

Ragnarok [Chr99c, Chr99b, Chr99a, Chr98b, Char98a] is a software development environment with focus on software architecture and architectural evolution. In Ragnarok, a document represents a software abstraction in a software system. A document may have one C-node only, and multiple N- and L-nodes. N-nodes store the implementation of the abstraction (source code), and L-nodes architectural relations (like composition, depend-on (import) or subclass-of) between abstractions. Ragnarok simulates composition using reference semantics (L-node links) and the tree-structure requirement is ensured by checking at the user interface level.

Ragnarok uses a traditional repository/workspace model. A session takes place locally in a workspace, and ended (changes are committed back to repository) by a check-in operation. Ragnarok has transitive change propagation over L-nodes. Thus, if a document, A, is changed then any document that includes A in its transitive, reflexive, closure of L-links is considered changed; but only locally in the workspace where the change was made. Ragnarok creates new, local, copies of all affected nodes and rebinds L-nodes to reflect the changed architecture.[1]

The session concept is highly flexible; essentially each document has its own session. A document's session is started by the first change to the document, directly (edit of N- or L-nodes) or indirectly (something in its transitive closure changed). A document's session is terminated by a check-in; and the check-in is propagated to all documents in its transitive closure. Thus, changes are committed to the repository and all sessions closed in the sub-configuration that is rooted in the document. However, document sessions higher in the hierarchy (documents not in the closure of the document, but related to the document) remains open, which is how version concentration is made in Ragnarok. As a concrete example, less than 30 versions of the root document in the ConSys system (see data below) was made over a two year period where the system's size more than tripled in terms of KLOC.

Finally, Ragnarok allows new configurations to be constructed intensionally in a workspace, as it provides a rule-based check-out. An example is given in section (4.4 Supporting concurrent work)

Ragnarok is currently used in three real development projects, outlined in the table below and detailed in [Chr98a]:

	ConSys	*BETA Compiler*	*Ragnarok*
Used since	Mar. 96	Feb. 97	Feb. 96
No. developers	3	4	1
No. files	1340	290	160
No. lines (KLOC)	240+binary	120	45

3.3 Summary

In this chapter we have presented the Unified Extensional Versioning model and three systems that use the model. The model improves on the observations regarding traditional models that we mentioned above.

[1] This propagation and rebinding mechanism simulates ordinary development where module relations are inherently generic: "A imports B" and thus any change in B indirectly affects A.

- Representation of configurations is direct. A configuration can be represented with a document that contains links to the other documents included in the configuration.
- Configurations are versioned. As any other document a configuration exists in versions. Versions of configurations are explicit, they can be named and organized.
- Versions of configurations are related to each other so their development can be traced. They can be compared and differences can be presented as components being added, deleted or changed. There is no need for auxiliary support such as 'Tagging'.
- Consistency is provided in the versioning sense. A version of a configuration can always be reproduced in exactly the same form. There is no need to copy systems in order to provide reproducibility.

In the presentation of the three systems we have highlighted how the model can be used and tailored to three common, but rather different situations: in a CM tool supporting software development in a traditional Unix tool-based setup, in an integrated tool for authoring papers, and in an integrated environment providing synchronous and asynchronous interaction for development in a geographically distributed setting. The Unified model go beyond traditional models in that it provide more support in a number of important situations.

- Version concentration. The number of versions of a configuration that has to be considered is greatly reduced compared to the possible combinations given by mathematics.
- Architectural traceability. From any level of configurations the exact changes that has been made over time, can be traced down to the individual file.
- Modularization. Configurations can be handled as modules where the internals and its detailed development is separated from its interface and its development from external point of view.
- Scaleability. Configurations can be included as elements in larger configurations thus forming hierarchies is directly supported. This is an essential property when managing any complex system.

These and other aspects of the model will be further discussed in the next chapter.

4 Discussion and Comparison

In this chapter we will discuss some effects and consequences of the unified extensional versioning model and its use and compare with the intentional model.

4.1 The Unified Extensional Versioning Model from the users perspective

A consequence of the unified extensional model is that the concepts ‘versioned component’ and ‘bound configuration’ are unified. Extensional versioning is used in both cases which means that the user can use the same model for versioning components as well as for versioning configurations. In the same way as a user can decide what changes go into a new version of a component s\he can control through the session mechanism what goes into a new version of a configuration. In both cases the version represents what the user regards as a meaningful state. The versions of configurations, including content and structure, are explicitly represented in the version database. This allows the user to identify, inspect, compare and reason about the properties of the configurations both in terms of content and structure: How and when new sections or chapters have been added or removed, how the dependency structure between software modules have evolved, etc. The hierarchical formulation of the model allows the user to organize the system in layers of libraries, sub-systems and systems all explicitly represented and versioned.

In a software engineering context, the extensional model implies that a version of a module not only embodies the source of that module but also contains information about the modules that it depends upon, which can be characterized as the SCM equivalent of the modularization principle. The developer creates what s\he thinks are meaningful and consistent combinations of versions of the included documents. The user of such a configuration (a library, module etc.), who have less insight in its internals, are thus confronted with choosing among a small number of meaningful versions of its configuration.

Builds of a system is always made from a bound configuration which in the extensional model is explicitly available as a version of the system configuration. Likewise, bill-of-material facilities are directly supported since the structure of the system and version of all components are given from a version of the system. What remains to capture is external aspects such as versions of used tools, options, etc.

In comparison the intentional versioning scheme is more complex from a user point of view. In order to specify configurations the user needs to master a separate selection mechanism for versions of configurations, often a small, specialized, language. (Languages that are often error-prone to use and does not deal gracefully with structural changes.) Encapsulation is weak since selection is performed over entire systems, also over parts not known in detail by the developer. Resulting, bound, configurations can be labled, but there is no support for comparing or relating such configurations to each other. As a result users are directed to produce and store listings of components and their versions in order to support bill-of-material facilities.

4.2 Managing the combinatorical explosion of configurations

The problem of combinatorical explosion is one of the fundamental problems which has to be countered in every model. In the extensional model this is achieved through the effect called ‘version concentration’. Consider first the tiny example in Figure 5c. On the document level, in D3 there are 2**3=8 possible configurations of versioned nodes of which only 3 have been created. On the relation level there are 2*3=6 possible configurations of the existing versions of D2 and D3, but here only 2 have been created. The fact that mathematical combinatorics give that there are in all 32 possible combinations of the versions of the leaf nodes in this small example is thus of no interest since the user have control over which combinations to explore and only these, for him/her interesting configurations, are created. Furthermore, the two-session update of D3 is only reflected as one new version of the configuration, D1. The hierarchical structuring in combination with the session mechanism is thus helpful in reducing the number of versions of configurations - version concentration also on the configuration level. For the rest of the system, using D1, the number of combinations of the files in this sub-system that needs to be considered is thus decreased from 32 to 2. Should, however, a user want to use another configuration of D1, say using the middle version of D3, the model makes it easy to represent such a configuration as another version of D1.

In realistic situations the numbers are much higher, 100 files in 10 versions each result in 10*100 mathematically possible combinations which are concentrated to perhaps 100 interesting versions of the configuration. Of these only a small number are relevant at any given time, often the last in each sequence of versions resulting from concurrent work (branch).The version concentration mechanism works in the same way at each level of configuring sub-systems into larger sub-systems and so on. At the system level there are comparatively few versions of the configuration corresponding to interesting versions of the system as a whole; releases, test-versions and so on.

In the intentional model the problem of combinatorical explosion is countered by using selection rules, ideally choosing the intended version of each file. Such rules are not directly depending on the number of revisions of files (i.e. the age of the system) which makes this approach scale up over time. The rules do, however, depend on the size of the system since the number of modules, each with its branches and labled configurations, will grow with the system. Selection rules are global and need to reflect all the modules at the same time. In contrast the hierarchical composition used in the extensional system scales well as illustrated with the Ragnarok experience. A system with 1340 files resulted in only 30 versions on the system level during a period of 2 years. A period when the system was heavily modified and trippled in size and the number of possible configurations would be uncountable.

4.3 Supporting and managing changes

A CM system must support simple and low-overhead facilities for developers to change and extend a system. Ideally such support should be possible to offer staying within the used versioning model. The main mechanism in the intentional model for this is generic selection rules, such as ‘Latest’, selecting the latest created revision of a modified file, which often is what the user intends to use. A configuration specification using generic rules will not need to be changed in order to include a new revision of yet another updated file and is thus convenient to use for a developer.

The corresponding mechanism in the extensional model is the session mechanism which allows several changes to a component as well as to a configuration to be included in one revision. Using this mechanism the developer will create a new revision of a component (or configuration) indicating that this part of the system is under revision. All changes the user makes to the component in this revision will be accumulated. When the user so decides the session is concluded and the version of the component is closed and can no longer be modified. When dealing with components, the situation in the extensional and intentional models for the developer comes fairly close. Check-out and check-in corresponds to creating and closing a revision of a component.

When dealing with configurations the situation is, however, different. In the extensional model the user needs to create revisions also of configurations in order to include revisions of its components, thus also if the component itself is not explicitly revised. Thanks to the session mechanism, the user can leave a revision of a configuration open and thus accumulate revisions of several of its components and also several revisions of the same component. Again, when the user so decides, the session is concluded and the user can thus control the granularity of the revision, for example to let a revision of a configuration represent a logical change. The experience from the use of Ragnarok shows that sessions tend to be longer the higher up the hierarchy the component is, and thus very long on the system level.

There are situations where a number of revisions needs to be created or closed at the same time. When the user decides to finish a session and close a revision of a configuration, all open revisions of its components that it uses must also be closed in order to form a bound configuration. This could be a tedious operation, involving many components. The Ragnarok system has demonstrated how it can find and close the relevant revisions of the components leaving to the user only to close a revision of the configuration acting as the root in a sub-graph. The users of Ragnarok has been interviewed [Chr98a] and they state that the ‘intermediate’ versions created were not problematic. ‘It is the job of the tool’ to handle the internal, possibly complicated, bindings, but the tool was reported to handle this adequately, and they did not find the presence of intermediate versions a problem. The ‘intermediate’ versions are, however, essential in order to facilitate full traceability in all situations. In the intentional model this operation corresponds to checking in components, labeling the configuration, and updating the selection rules (making sure generic rules are replaced), seemingly a heavier operation.

The extensional model trivially supports reconstruction of a version of a configuration that has been closed since it can no longer be modified. In the intentional model this takes a correctly formulated, and stored, set of selection rules, which is hard to guarantee in particular in presence of heavy restructuring of the system. Alternatively one has to store the full list of components and versions for the entire system. On top of this the extensional model offers full traceability among the explicitly stored versions of configurations. It supports relations between such versions of configurations and a tool can show how they are derived from each other, compare them, show the differences down to every included component.

4.4 Supporting concurrent work

In projects involving many developers it is often a necessity that work can be done concurrently by several developers, including revising the same documents and configurations. To make this a practical possibility, it must be simple and swift to merge the result of concurrent work affecting both the component and configuration level. Merging concurrently developed revisions, temporary variants, of a component is an established technique. Here tools make use of the known content of the two temporary variants and their common ancestor to perform a three-way-merge, suggesting the resulting merge and detecting lexically interfering changes in the two variants. Dealing with configurations the work is often structured so development starts from a common alternative, but done in a separate alternative. When such a task is concluded the revisions are made available by updating the common alterative. In case of concurrent work, any changes in the common alterative must first be merged with the new changes in the separate alterative, tested etc., and then used to update the common alternative. Thus the last developer to conclude his concurrent work will have to deal with merging with earlier work.

In the intentional model concurrent work is often aided by workspace areas where the revisions of changed files are stored and visible for the local developer. The tool then aides in updating the common alterative as well as merging parallel work, i.e. updating the workspace with files changed in the common alterative and initiating merge of files that has been changed in both places.

In the extensional model configurations are explicitly versioned and concurrent work is represented as variants in its versiongraph. Merge is thus achieved in the same way as for components - a new version is created with the variants as predecessors. With the same rules as in the intentional system a tool will select the latest revision of a component changed in only one of the alternatives and initiate a merge of a component that has been modified in both alteratives. Since the model is recursive a component might be a new configuration and the process repeated until all components have been merged (the same ones as in the intentional model), and the affected configurations have been facilitated with a new version representing the merge. The difference between the models thus lies in the last point. The explicit versioning of configurations makes it simple to explore the history of configurations which is particularly useful in the context of concurrent work and merges.

In the merge-case above we notice that all the versions of the involved configurations are a consequence of the model and can be automatically managed by a tool. A similar situation occurs when one want to integrate with changes to the system unrelated to the concurrent development. In the intentional model this is provided through the generic rules (e.g. the 'latest' rule of ClearCase, and the CVS command 'cvs update'). As an example of similar functionality in the extensional model, the Ragnarok prototype provides a command, 'gettip', that specifies that the latest version of any component should be used in the users workspace. This command retrieves the latest revision of all components from the version database, updates the bindings between the components and configurations in the workspace, creating new versions of configurations as needed. This is a proven and often used technique to merge parallel work of different parts of a software system.

4.5 Implementation aspects and some usage experience

Storage space overhead is an important aspect when managing large systems. When storing components, standard delta storage techniques can be used as usual for compact storage of revisions. On top of that, the model presented in this paper can represent internal structure in a document, which can be used to share common nodes and subtrees between variants facilitating compact storage and fast retrieval of variants. The representation of bindings between documents, L-nodes, is comparable to what is already present in form of external declarations (or comparable mechanisms) in source-files. The representation of explicit versions of these bindings is an additional, but very small cost and to store differences of these bindings is very compact. It should be compared to label all files in a system using the traditional approach. Although we have not made a careful study of this we are confident that our approach will come out favorable in a comparison due to the hierarchical structure (even if the labels are chosen very short).

In all long-lived systems the version history becomes long-winded and partially uninteresting. In particular long sequences of successive updates tend to be of little interest after a while. This is a general problem that can be observed already with common tools for versioning components. In the extensional model the effect of 'intermediate' versions may contribute to make such sequences for configurations even longer. In any case the problem is general and in a graphical interface (such the one used by some of the systems described earlier) may have to consider techniques where such sequences are collapsed, but still accessible, in the presentation.

In this paper we have argued that the 'version concentration' effect will eliminate the potential overhead created by the intermediate versions of configurations. This effect is created by the session mechanism (collecting revisions over time) and the version propagation mechanism (collecting revisions over a sub-tree/sub-graph of the system). The last mechanism will work better with a certain fan out at each configuration and will clearly not help in the unlikely situation of a system built as a linear list of components. In order to see how 'version concentration' worked in practice, the Ragnarok prototype was in early February 1997 equipped with two additional house-keeping attributes, that allows the actual amount of proliferation in the version database to be assessed quantitatively. During check-in each new version stores two boolean values: 1) if this version has a change in the 'local data' attribute compared to the ancestor version, and 2) if this version was created as a result of a directly issued check-in command. In Figure 6 below, data for these attributes is shown for every quarter of the year in the period 1997 to 1998. Column T shows the total number of versions entered into the version database during the indicated period. S is the percentage of T where attribute 1) is true, i.e. the percentage of versions where 'local data' was changed. Similarly, O is the percentage of T where attribute 2) is true, i.e. the percentage of versions created by a direct command. Thus 100%-S is the percentage of intermediate version, versions that would not have been created in an intentional model.

The important point is the stability over time of the percentages O and S. The number of version nodes in the repository is proportional to the number of check-ins and to the number of changes; thus there is no combinatorial explosion. Furthermore, the numbers tells us that there is roughly one 'intermediate' version for each 'essential' version. For each explicit check-in there is 3-8 files checked in (which means 1.5-4 'essential' versions). Thus rather than creating more work for the user having to check

	ConSys			*BETA Compiler*			*Ragnarok*		
Quarter	*T*	*O*	*S*	*T*	*O*	*S*	*T*	*O*	*S*
1997 I	-	-	-	-	-	-	13%	12%	44%
1997 II	405	20%	29%	255	34%	53%	447	13%	36%
1997 III	332	43%	58%	505	30%	55%	457	12%	39%
1997 IV	290	24%	33%	366	23%	62%	138	11%	32%
1998 I	289	31%	43%	253	34%	66%	111	11%	54%
1998 II	499	25%	32%	624	29%	64%	106	12%	36%
1998 III	478	44%	51%	385	26%	52%	207	10%	35%
1998 IV	349	19%	46%	147	32%	60%	73	13%	57%
Sum	2438	32%	45%	2518	30%	60%	1648	13%	40%

Fig. 6. Data from use of Ragnarok

in 'intermediate' versions the situation is that in Ragnarok a user have to handle fewer explicit check-ins than in a traditional system.

4.6 Support for variant selection

The presentation of the Unified Extensional Versioning Model in this paper has focused on its support for versioning, including temporary variants for concurrent work. The presentation has not considered support for permanent variants. The intentional model has an advantage here in that its selection rules can be used both to select among revisions and variants. The extensional model thus needs to be extended in order to support representation and selection of variants. The authors do, however, feel that an interesting approach would be to include facilities in the tradition of 'conditional compilation' and thus provide conditional parts of a document. This would make it possible to keep variant parts close together, often preferred by developers, rather than enforcing them to be separate files (or separate variants of a file). Integrating support for variants in the model is for the time being left as future work.

5 Conclusions

In this paper we have described a new model for versioning of both components and structure of a system. In our analysis of existing approaches we came to the conclusion that in general it is better to use the same concept for the versioning of structures as for the versioning of components. Furthermore, we identified some weak points in the intentional model for versioning of structures and that it had some weaknesses compared to the extensional model used for atomic entities. This led us to propose our unified extensional versioning model using extensional versioning for both atomic entities and configurations. We have shown that the good aspects of the intentional model, flexibility in development and reducing the problems of combinatorical explosion, can also be achieved in the extensional model. On top of that we also have shown that it facilitates a number of other essential aspects. The applicability and use of the model have been demonstrated by presenting three different systems built using the model.

We have also shown that the unified model is superior in several important aspects.

- It offers the Version Concentration mechanism to counter the combinatorical explosion of versions.
- The explicit representation of Configurations makes it possible to organize configurations hierarchical, and to modularized configuration management,
- The explicit versioning of configurations makes it possible to guarantee repeatability, a version of a configuration is well defined and can always be recreated. This makes it unnecessary to make dedicated copies of important configurations - such as releases. It support architectural traceability which is achieved through comparing versions of configurations, and it also directly support grouping related changes together into logical changes.
- The model is general since it does not restrict the tool-implementor and the model in itself does not impose any specific policy or process on the user.

The use of the model in three systems for rather different situations support the claims on generality and of cause also of the usefulness of the model. Experience and data from in particular one of the implementations support the claim that version concentration works also in practice.

On a more subjective level we think it is a benefit that the problem of managing configurations and atomic entities can be handled through one set of concepts and mechanisms rather than two. Rather obviously we find the concepts we have developed natural and simple, but more importantly we have found them easy to explain and to adapt by users. This makes us believe that the model we have presented is to some extent 'natural' and since it also powerful, as argued above, we like to suggest it as an alterative for others building CM systems.

References

[AdC90] Software Maintenance and Development Systems. Aide-de-Camp Product Overview. Software Maintenance and Development Systems, Concord, MA 1990.

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

[Ask99] Ulf Asklund. Configuration Management for Distributed Development - Practice and Needs. Licentiate thesis, Dept. of Computer Science, Lund University, Sweden. 1999.

[AM97] U. Asklund and B. Magnusson. A Case-Study of Configuration Management with ClearCase in an Industrial Environment. In Proceedings from SCM7 - International Workshop on Software Configuration Management, R. Conradi (Ed.), Boston, May 1997, LNCS, Springer Verlag.

[AMP99] U. Asklund, B. Magnusson, and A. Persson. Experiences; Distributed Development and Software Configuration Management. In *Proceedings from SCM9 - International Symposium on System Configuration Management*, J. Estublier (Ed.), Toulousem France, Sept. 1999, LNCS, Springer Verlag. To appear.

[BLNP97] Lars Bendix, Per N. Larsen, Anders I. Nielsen, Jesper L. S. 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] Lars Bendix, Per N. Larsen, Anders I. Nielsen, and Jesper L. S. Petersen. CoEd - A Tool for Versioning of Hierarchical Documents. In Magnusson [Mag98].

[Ced93] Per Cederqvist. Version Management with CVS. Available from infosignum.se, 1993.

[Chr98a] Henrik Bærbak Christensen. Experiences with Architectural Software Configuration Management in Ragnarok. In Magnusson [Mag98].

[Chr98b] Henrik Bærbak Christensen. Utilising a Geographic Space Metaphor in a Software Development Environment. In Prasun Dewan, editor, *Proceedings of EHCI'98, IFIP Working Conference on Engineering for Human-Computer Interaction*, Crete, Greece, September 1998. Kluwer. To appear.

[Chr99a] Henrik Bærbak Christensen. The Ragnarok Architectural Software Configuration Management Model. In Jr. Ralph H. Sprague, editor, *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1999.

[Chr99b] Henrik Bærbak Christensen. The Ragnarok Software Development Environment. *Nordic Journal of Computing*, 6(1), Jan 1999.

[Chr99c] Henrik Bærbak Christensen. RAGNAROK: An Architecture Based Software Development Environment. PhD thesis, Department of Computer Science, University of Aarhus, Denmark. 1999.

[Clear] http://www.rational.com/products/clearcase

[Crn92] R. D. Cronk. Tributaries and deltas. BYTE, pages 177-186, January 1992.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232--282, June 1998.

[GKY91] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in EPOS. Soft. Eng. J. 6, 6 (Nov.), 378-386. 1991.

[HM88] G. Hedin and B. Magnusson. The Mjölner 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.

[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.

[MA96] Boris Magnusson and Ulf Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In Sommerville, I., editor, Proceedings of the 6th International Workshop on Software Configuration Management, LNCS, Springer Verlag, Berlin. 1996

[Mag98] Boris Magnusson, editor. *System Configuration Management*, Lecture Notes in Computer Science 1439. ECOOP'98 SCM-8 Symposium, Springer Verlag, 1998.

[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] Magnusson, Hedin, and Minör (eds). Proceedings of Noridc Workshop on Programming Environment Research. Lund, June, 1994.

[MLG$^+$93] B.P. Munch, J.-O. Larsen, B. Gulla, et. al.. Uniform versioning: The change-oriented model. In Proceedings of the 4th International Workshop on Software Configuratino Management. Baltimore, MD, May 1993.

[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.

[MMAxx] Boris Magnusson, Sten Minör and Ulf Asklund: A Model for Semi-(a)Synchronous Collaborative Editing. To appear in Journal of Computer Supported Collaborative Work.

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

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