A Case-Study of Configuration Management with
ClearCase in an Industrial Environment

Ulf Asklund and Boris Magnusson

LU-CS-TR:97-184

Also published in: Proceedings of SCM7, International Workshop on Software Configuration Management, R. Conradi (Ed.), Boston, May 1997, LNCS, Springer Verlag

Department of Computer Science
Lund Institute of Technology
Lund University

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

A Case-Study of Configuration Management with ClearCase in an Industrial Environment

Ulf Asklund and Boris Magnusson

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

Abstract: This paper reports from a case study where the configuration management system ClearCase is used in a large scale industrial application. The focus of the study is on the functionality offered and how it matches the needs in this particular situation. The paper reports on situations where ClearCase has turned out to be cumbersome to use or is lacking functionality. Improvements are outlined for how the problems can be solved or the situations better supported. The suggested improvements are influenced by experience with the COOP/Orm research prototype and some of the improvements are illustrated with functionality available in this integrated environment.

1 Introduction

Configuration management is a well-known problem in industry since most products evolve over time. There is a need to keep track of which components are included in a specific version of a product. The car industry is a typical example where each car model is often revised every year and versions of spare parts for older versions of a model needs to be identified during a long time period.

The situation in the software industry is at first sight similar, but it turns out to be much harder to cope with. Explanations for the increased difficulties can be sought in aspects such as a much faster change of versions (including internal versions), less visibility of incompatibility (compared to mechanical parts), very complex systems, and less standardization of parts and sub-systems. In academia and in parts of the software industry these problems have been identified and systems for management of software components have been developed. A first generation of tools, such as SCCS [Roe75], and RCS [Tic85], were based on version control of single files. These systems give limited support, but are simple to introduce and to use. A second generation of tools, which take a broader view on support in the software development process, such as Continuus/CM [Cont, Cla95], ClearCase [Clear, Cla95], and Teamware [Team], are now finding their way into industry. While giving more support they also have a larger influence on the developers´ work process and are thus harder to introduce. The acceptance of such tools is a comparably slow process and seems to have taken place mainly in some kernel software industry. Many industries are now, however, facing the situation that software components are part of what they used to think of as mechanical or electronic products. As the software components grow, the traditional methods for version control and configuration management may prove inadequate and there will be a need for more sophisticated software configuration management tools in a wider community. At the same time there are research activities to produce even more sophisticated environments and tools. There is thus an interesting question of to what extent the existing systems meet the needs for configuration management support, in the pure software industry as well as in the traditional industry.

Introducing configuration management tools in an existing organization is not always easy. Such tools inflict some overhead on the developers in their everyday short-term work, which they may not recognize as motivated. The benefits seems to be more visible for managers, and in the long run. It therefore often takes an explicit management decision to introduce configuration management in an existing organization. The needed understanding of CM problems, principles, and systems is more likely to occur in large-scale software companies, but the need for CM is equally important in smaller companies. In introducing configuration management tools it is important that the tools are appreciated as a help rather than as an additional burden for the developers. They must thus be easy to use and understandable in terms of the development process used at the company. In a large company there might be resources to create a specialized support group for configuration management, but in smaller companies these tasks must be handled by the developers themselves. There is thus also a demand that the tools and systems must be easy to manage. Tasks like creating new development lines, branches, merging and integration of development lines, finding out what development lines exists, etc. must be possible to do without too much training and work.

In this paper we report from the use of a second generation system, ClearCase, in a software company. We have been particularly interested in the functionality it offers and how it fits the needs of the company. The aim of this case study was to do an objective, independent and critical examination of how a company with well developed configuration management handled their software. We have tried to identify shortcomings and suggest solutions of how they can be overcome in the current framework. We have also compared the system and the situation with the research prototype we are working with, COOP/Orm. The goal here is to understand how the needs at the company can be supported in order not to miss important aspects of real use. We outline how the model used in the COOP/Orm system can go further in the support offered, in particular in the areas of providing group awareness, fine grained version control and supporting synchronous interaction.

In section 2 we give some background on the company, what they do and the environment they work in, a short introduction to ClearCase and to the COOP/Orm prototype. In section 3 we describe in more detail how ClearCase is used today, including local adaptation. In section 4 we identify and describe particular problems with the use of ClearCase. In section 5 we outline how the same situation could be handled in an environment with the facilities available in COOP/Orm. In section 6 we outline solutions of how the problems could be solved or reduced in the ClearCase framework. Section 7 discusses directions for future work, and section 8 concludes the paper.

2 Background

2.1 Introduction to ClearCase

ClearCase by Atria is a version-control and configuration-management system, designed for development teams working in Unix on a local area network. ClearCase MultiSite extends the ClearCase features to also support geographically distributed development teams. ClearCase stores all data under its control in versioned-object bases (VOBs). A VOB is an implementation of the NFS [NFS] protocol and access can thus be physically distributed in a local area network. VOBs also inherit from NFS the restrictions that they can not span physical discs. Each file and directory visible through a NFS-mounted VOB appears as a Unix file or directory which gives a high degree of transparency to standard tools. A VOB can be replicated to remote sites, but there remains a notion of ownership with one of the sites.

Versioning is always in the context of a particular *branch-type*. A branch-type is identified with a name and is relevant for all files in a VOB. Initially there is only one branch-type, 'main', but at any point in the development a new branch-type can be created. There is no assumption of any relation between branch-types in ClearCase. A file can exist in a sequence of versions on a branch-type (often called a '*branch*' of the file). A new version of a file in a particular branch-type is created by 'checking out' the file. A file can be checked out, exist, on several different branch-types at the same time. Synchronization within a branch-type is by locking, preventing developers working on the same branch-type to simultaneously change the same file. Versions of a file on different branch-types can, however, be checked-out in parallel.

All users access versioned source data in the VOB through a *view*, and a virtual *workspace*. A View provides a selection mechanism and a workspace is a storage area (directory) in which developers can perform tasks in isolation from other development. Each view has an associated *configuration specification* which lists rules for selecting a version of each needed file. Many rules name a branch-type on which to look for a version of a needed file. The rules are evaluated in order until a rule matching an existing version of the file is found. They can thus be used to specify an order among the branch-types to use in the particular view. Each rule can be dynamic, allowing users to see, for example, the latest version of a file on a branch-type, or it can be fixed to allow a developer to work with a fixed version of a file regardless if later versions of the file exists on that branch-type, or it can simply name a particular version of a file or a 'label' (as of below).

A development project typically use a particular branch-type and a view. Files that have been changed in the project have versions on that branch-type. These versions are often thought of as a branch (variant) of the file. The view specifies a dynamic rule to select files that have versions on the branch-type, perhaps in several steps, and at the end some fixed rule, which is used as default to get a stable version (such as the latest release) of files that have not been changed in the current development project. Views that only contain fixed rules will always return the same version of all files and can thus be used to represent a version of a configuration, a baseline.

Merging of branches is done using the ClearCase merge tool. It identifies the differences between the branch-types to merge in terms of files that have been updated on the branch-types. For each of these files it identifies a 'common ancestor' version of the versions used on the branch-types. Files that are changed in only one branch are included in the updated version. Files that have been changed in more than one branch, but where the changes do not effect the same original source line, can be automatically merged. For files with changes on the same source line, the developer is prompted to choose which changes to accept in order to resolve the conflict.

Views are used to select a configuration of a system, but since this selection is done on demand it gives different results as new versions of files are created. In order to make it possible to come back to a particular set of versions of files there is a need to identify versions of configurations. This is done through *labeling* - all the versions of files in such a set are marked with the same label (such as "Release.2"). A view can use labels to select versions of files to recreate the configuration. Although views and thus branch-types can be used to name a labeled configuration, there is afterwards no relation between the label and the branch-type or indeed between the labeled configurations. Branch-types are thus only a kind of common naming for variants of files, but do not support organizing versions of configurations.

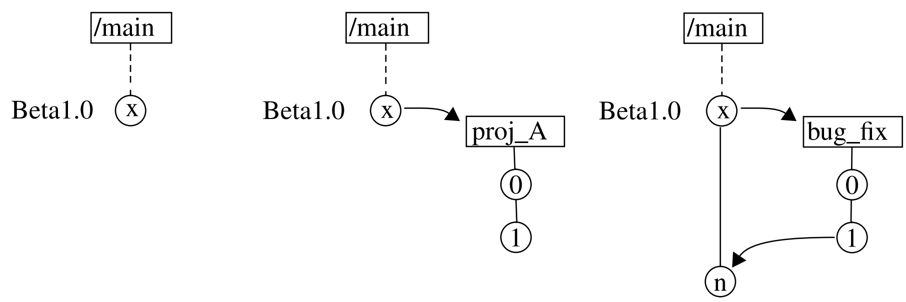
For replicated VOBs, branch-types come with a protection mechanism which restrict creation of versions of files on a particular branch-type to one site. Versions of files on the built in branch-type, 'main', can only be created on the site that acts as the master of the VOB.

A triggering mechanism can be used to extend the functionality of ClearCase by scripts executed at situations such as check-out. This facility can be used for all kinds of extensions such as logging, restricted access, work process, etc.

Small example

We will with a small example further illustrate the functionality of ClearCase. The example also describes a common work process of how branch-types are used.

Consider a small system which consists of ten files of source code, named foo1.c to foo10.c, all stored in the same directory. In the initial situation the program has been



a) Unchanged files

b) Files modified on branch-type 'proj_A'

c) Files modified on branch-type bug_fix and then merged to main

Figure 1. Version trees for individual files

developed within ClearCase, but without using any branch-types and that the latest version of all files are labeled with the label 'Beta1.0'. I.e. all files, including the directory, have all been versioned on 'main', where they may have reached different version numbers. Thus, each file has a version tree as depicted in Figure 1a, where x is the latest version for that file. The next step in the development of this program is to implement new functionality but also to correct bugs reported from the beta testers. These two tasks should, by the two developers Ulf and Boris, be accomplished in parallel and their individual results should then later be merged and labeled forming the first release.

For the purpose two branch-types, proj_A and bug_fix, are created. Ulf, who is responsible for the new development, has the following configuration specification in his view:

```
element * CHECKEDOUT
element * .../proj_A/LATEST
element * /main/Beta1.0    -mkbranch proj_A
```

When a file is needed, the rules are evaluated top-down until a matching version of the file is found. A rule, i.e. a line in the description, consists of four parts: type of item evaluated by the rule, restriction on files to be evaluated due to their place in the structure, the rule, and (optional) the branch that should be created when a file chosen by this rule is checked-out. The rule itself can contain either a specific version, a label, or one of the predefined functions: 'CHECKEDOUT' or 'LATEST'. The first rule will select a checked out version of a file. In case there is no such version, the second rule will select the latest update of the file on the branch-type 'proj_A'. Finally, for files that have no version matching either of these, the version of the file labeled 'Beta1.0' will be selected.

Within the view Ulf implements the new functionality in isolation from Boris' changes. To accomplish his task foo1.c and foo2.c are checked-out, modified and checked-in. Thus the two files have a version tree as depicted in Figure 1b. The files and the directory not yet modified (if not Boris has created new versions of them) have the same version tree (Figure 1a). I.e. a branch is created on demand for each file. We call the creation of a branch, i.e. when, for a specific file, the first version on the branch is created, step A in the work process.

In parallel with Ulf's work Boris check-out, modify and check-in the files foo2.c and foo3.c to accomplish his task. When the bugs are fixed and tested the bug_fix branch-type is merged to main. Merges are also visible in the version tree, depicted by the file foo3.c version tree in Figure 1c.

Ulf just finished his task and is now ready to merge his changes to main. However, to avoid incorrect files on the main branch-type the work process convention is that integration tests must be made on the branch-type before merging to main. This means that before a merge to main is performed, the program should first be merged to the development branch-type and tested there, step B. Possible errors due to merge conflicts can in this way be corrected on the branch-type in isolation from all other projects.

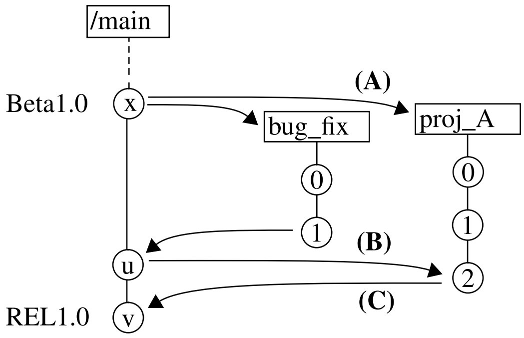


Figure 2. foo2.c after merge of two sets of parallel changes to main. The changes in the second set are first tested locally (B) before merged into main (C).

Before merging the configuration specification must be changed to select the latest version on main instead of the labeled one. I.e the specification must be changed to:

```
element * CHECKEDOUT
element * .../proj_A/LATEST
element * /main/LATEST   -mkbranch proj_A
```

When the program has been tested and checked-in it can now be merged to main, step C. The specification is again changed, now to:

```
element * CHECKEDOUT
element * /main/LATEST
```

and the merge is made. If no further versions have been created at the main branch-type since step B, this merge is trivial, otherwise step B should be repeated. After the merge to main (C) the program is again tested, files are checked-in, and the selected versions are labeled. By labeling the versions a new version of the entire system is created and can later easily be recreated.

Figure 2 depicts the version tree of the file foo2.c. Both Boris and Ulf modified this file and therefore versions of it exists both on branch-type bug_fix and proj_A. The work process is also depicted with its three steps: (A) the branch-type is created, (B) updating the branch-type from main, and (C) finally merging the branch-type to main. For this file all three steps are visible in the version tree. However, if, for a file, no new version had been checked-in between step A and step B, step B had not been visible in the version tree of that file.

Further details about ClearCase can be found on Atria's www-site [Clear]. ClearCase is also one of the configuration management tools evaluated by Ovum [RBI95].

2.2 Kockums Computer Systems

Kockums Computer Systems, KCS, markets design and production information systems specifically created for the shipbuilding industry. Systems from KCS are currently in service at more than 260 sites in 38 countries in Asia, Australia, Europe, North and South America. KCS' main office is located in Malmö, Sweden, with subsidiaries in Germany, Japan, South Korea, United Kingdom, and the USA.

Their product, TRIBON, is a CAD/CAM/CIM system and is in fact 17 different products combined to one complete system. TRIBON is ported to several hardware platforms including VAX, ALPHA, HP, IBM, Sun, and PC (partly). These platforms in turn have several operating systems in several versions, e.g. HP-UX 9, HP-UX 10, AIX 3, and AIX 4. The programming languages used are among others PL/1, Fortran, C, C++, UIL, and assembler. They, of course, also exists in several versions. These aspects in combination with a geographically distributed development with approximately 60 developers in three different countries makes it a complex development environment interesting to study. To handle this complex development situation KCS has had a long history of emphasis on configuration management resulting in a well developed CM-model, implemented in ClearCase and ClearCase MultiSite.

2.3 The COOP/Orm environment overview

COOP/Orm is an environment supporting collaborative writing of hierarchically structured documents. It is based on previous work in the Mjølner-project [KLMM93], concerning object-oriented software development. The environment developed in Lund, Mjølner Orm [MHM+90], supports collaborative software development to a limited extent through its configuration management [Gus90]. The current project, the COOP/Orm environment, is a research prototype aimed at providing support for distributed teams of developers. It has been influenced by advances in the CSCW area, but its main target area is software development.

COOP/Orm is designed for use in a distributed environment and is built with a multiple-client, multiple-server architecture. Some of the aspects of distribution and replication of data was reported at PDCS-96 [MG96] and in [MA95]. As a result of the design, users of COOP/Orm can have the same support and interaction whether local or distributed although the speed might depend on the quality of the long distance network.

COOP/Orm is also designed to support versioned configurations of documents (such as programs). Links from one document to another is always denoting a particular version of that document (which might be a new configuration). As a result, selecting a version of a document determines particular versions of all (recursively) referenced documents. The model thus support a baseline approach, but the user interface makes it very simple to switch to use another version of a configuration as described at last years SCM [MA96]. Versions of configurations can be interactively compared and differences presented (added/deleted dependencies, change of targeted version).

COOP/Orm supports hierarchical composition of documents in an integrated storage model as described in [MMA93]. This storage model is compact since unchanged parts (or sub-trees) can be shared among many versions. The model support very fast identification of changes in terms of structural changes and changes to the content of a node [Ask94]. The hierarchy can be used to support structured software as e.g. classes and methods (or modules and procedures), but fits equally well for documents with chapters, sections, paragraphs etc. The hierarchical model support automatic merge of variants. Changes such as addition, deletion or change of a part in only one of the variants are handled according to default rules. Parts that have been changed in both variants has to be dealt with by the editor for that particular kind of node (also here default rules are supported).

COOP/Orm maintains a version graph for each document which in the user interface can be used to (very fast) switch between viewing different versions of a document and to compare (also distant) versions. The version graph is shared among all users and is updated when any user creates a new version of the document.

For the full benefit of the COOP/Orm model editors for the leaf nodes (such as a text editor) must be version aware in order to support the fine-grained and interactive version control mechanism. As an alternative documents can be written to normal files, edited with some standard editor, and then read back in, at which time the changes will be visible to others.

In contrast to other version-control/configuration-management systems, COOP/Orm offers a high level of collaborative awareness. These aspects are described in more detail in [MM93] and [MMA97]. In the present setting we like to mention a few aspects.

The version graph makes it possible for every user to get an overview of how his version of a document relates to other versions, and to see as new revisions and variants are created by other users. He can also compare his version with other versions, including these under construction as an extreme case. The model also supports synchronous interaction, but that is not further elaborated here.

The support for versioned configurations makes the user aware of when new versions become available. Switching to use another version of a configuration is simple and explicitly visible, and it is as simple to switch back. Selecting a version of a configuration implies selecting versions of all files included in the configuration (possibly very many), and at the same time ensures that a meaningful collection of these files is used ('meaningful' in the sense of an intended combination of versions).

3 CM overview at KCS

3.1 The CM history in the company

Like almost every company the developers at KCS first built their own configuration management system. Within the system, called MMS, the product structure were made. By use of name-conventions, files with functional relationship were structured into a family stored in the same directory. The system had functionality to, by the file name, find the directory containing a specific file and present the header. It was also possible to, very quickly, find dependencies between files (a useful functionality not yet fully implemented in the new system). Until recently all KCS customers had VMS platforms, but Unix use grew and the demands on porting TRIBON to other platforms increased. Unfortunately MMS only run on VMS and KCS therefore started to look at commercial alternatives, of which ClearCase were chosen.

1994 the, well planned, conversion from MMS to ClearCase started. For about 6 months ClearCase was used only as a secondary system and development was still made using MMS. In this way the developers could learn the new system gradually and not necessarily all developers at the same time. Even when ClearCase had become the primary system, MMS was used as backup. Platforms were ported to ClearCase one at the time.

Because the conversion was made gradually and the old product structure remained there was not much resistance from the developers. In fact, the structure and the name conventions are still used.

3.2 How ClearCase is used today

All source code is under the control of ClearCase, i.e. both code that make up the product, and all internal routines and scripts. The source is organized in total 12 VOBs, due to the technical restriction that a VOB can not span physical discs. So far there is no logical or structural background to on which VOB which information is placed. The same file type is used for all code, independent of platform and programming language. Such information are instead obtained by structured comments in the file header. A CM support group has been formed in order to maintain specialized competence and off load the developers from CM tasks. The CM-group works with troubleshooting and there are particular actions that usual developers are not allowed to perform.

Branch-types are used at KCS for three basic purposes: (1) each development project has its own branch-type, (2) maintenance is done on product branch-types, (3) correction of serious urgent errors are made on a special bug-fix branch-type. Development projects are typically relatively long-lived activities involving several developers. New branch-types are created by the CM-group only, which also help in creating views.

A build of a system is managed through a preprocessor that with a given view extracts the code for a given platform and activates the corresponding compilers etc. Management of variants for different platforms are thus done outside the control of ClearCase, but it enables storage of all code in the same place, i.e. on the same VOB. E.g. are the same discs mounted both on VMS and Unix systems. This enables even VMS code to be under control of ClearCase (which is not ported to VMS). ClearCase is however not used to handle documentation. The reason for this is that ClearCase only fully supports ASCII files and not binaries like word processor files, since the used delta techniques and merge tools do only work on ASCII.

The CM-group has built a command interface on top of ClearCase in order to make it easier to use for the developers and to give additional support. One example of a command is 'cmmkelem' which creates a new file. Before the new file is created a check is made to check that a file with that name does not already exist in some other branch-type. ClearCase would allow this, but it would create problems during merge. The command also supports KCS conventions for naming and organization in directories. Emacs has also been extended to support CM functions.

4 Requirements and shortcomings of ClearCase

The analysis is made with focus on some key areas. In this section each of these areas is stated, and the way that particular problems are handled at KCS is described and reflected upon. A more elaborated suggestion to changes (improvements) is made in section 6.

4.1 The version selection problem for developers

The problem concerning the selection of files and directories to be included in the workspace and to determine which version of each of these that should be chosen is often called the *selection problem*. The problem increase with the number of files and directories. One core functionality of ClearCase is the configuration specification in a view. A specification filters out the correct version of every file and directory to the developer. The idea is that once the view is set everything should almost be like 'normal', i.e. like working alone. 'Set and forget' is the slogan used in ClearCase manuals. The specification actually serves two roles: define the selection rules, and define the actions to be taken when a file is checked-out. Thus, a configuration specification is very powerful. It seeks through an amount of files and directories a developer hardly can get an overview of and comes up with the result. One danger, however, with this kind of global search approach is that as long as the specification is correct nobody care about how it works, but when then an error occur (which, sooner or later, always happens) it can be hard to solve, or even to notice.

The most frequently encountered CM-related problem at KCS is that the view specification is wrong. Even if the syntax is easy to understand the specification gets complex and it can be hard to see the consequences of a written specification. I.e. it is hard both to verify that a given specification is correct, but also to notice when there is an error. A misspelling, for example, can lead to that completely wrong versions are seen and used for a long time without notice. Even if errors are not so frequent at KCS, they have resulted in a suggestion that only the CM-group should create view-specifications, instead of the developers themselves as today. This will, of course, reduce the CM-knowledge of the developers even more. An observation is that a semantic check of the specifications, finding errors like misspelled branch-type names should reduce the number of errors. However, even with such an improvement the more serious problems of missed, or specifying the wrong, branch-type still remains. A better overview of how the branch-types relate to each other might help, but this is not supported by ClearCase at all.

More support for structuring could make it possible for the developers to get an overview so that they, by themselves, can do the selection or, at least, verify that the selection automatically made is correct. ClearCase supports structure through directories. Versioning of directories is only related to which files are included in a directory, not which version of these files. For decreasing the complexity in the version selection problem ClearCase directories thus offer no help.

Summary – needed support:

- Semantic check of view-specifications
- Support for relations between branch-types
- Support for versioned sub-systems (i.e. directories)

4.2 Support for short term sub-project

At KCS development is divided into several projects. Each project has its own branch-type which makes it possible for developers in different projects to work in parallel. In addition there is a branch-type dedicated to bug fixes. Branch-types are created by the CM-group, but any developer can create a version of a file on a branch-type. All developers working on the same project are working on the same branch-type which makes it important to know how their changes are shared. There are two fundamental strategies: *optimistic* where changes are shared as soon as they take place, and *pessimistic* where changes are seen by others only after their own consent. In ClearCase the strategy is controlled by the views as we will see below.

There are two basic problems elaborated upon in this section: how and by whom branch-types are created, and how to support optimistic and pessimistic propagation of changes within projects.

Strategies for branch type creation

At KCS creating a branch-type is viewed as creating a new project, which is nothing a developer does, but is restricted to be made only by the CM-group (having consulted the product managers). The CM-group thus maintains the knowledge of which branch-types that exist and for what purposes. The drawback is the extra overhead for the CM-group that runs the risk of becoming a bottleneck, and that the developers perceive creating a new branch-type as a heavy operation. Even with these restrictions there are many branch-types and a risk that merge problems occur due to related changes made in different branch-types. Synchronization within a branch-type is obtained by locking – it is thus impossible for developers working on the same project to work in parallel on the same file (although this is possible between project with their own branch-types).

There is a need for developers in the same project to work independently for shorter times. This could in principle be supported by using branch-types, but because of the management overhead on creation of branch-types, motivated by its main use, this would be too heavy-weight.

Baseline vs. generic configurations

A *baseline* means (following the terminology of Tichy [Tic88]) a configuration that specifies the versions of all included files and directories. A *generic configuration* on the other hand is dynamic and is set up using rules rather than fixed versions. Both these strategies can be supported in ClearCase views. Generic configurations are supported with the built in rules ‘CHECKEDOUT’ (matching the versions in the workspace) and ‘LATEST’ (matching the latest version of a file in a branch-type). Baselines are supported by using fixed versions and labels as rules.

Returning to the first configuration specification in the example in section 2.1, we can see how a view can be written to specify an optimistic change propagation strategy.

For developers working in the same project there seems to be at least three levels between using a baseline and a generic configuration for all files.

1) The most generic configuration is when the developers use the *same view.* This results in all having exactly the same workspace, i.e. even the same checked-out files are used although only one developer can actually change a checked-out file.

2) 'Normal' generic configuration is obtained by using *private views,* but with the rule to select the latest version in the project branch-type (this case is illustrated in the example view in 2.1). In this case all checked-out files are private and they get visible to (and are used by) other developers in the project when checked-in.

3) Finally, it is possible to work in a baseline configuration, which is obtained by having each developer work alone on a *private branch* (or have a private view with rules selecting specific versions). In this case changes made by other developers will not be incorporated without the user changing his view.

The positive aspect of working in a baseline configuration is that the developer has a stable environment, all differences in behavior of the developed system are related to changes he/she makes. The disadvantage of using baselines during development can be that changes are integrated late and get less tested together with the other developers changes.

Generic configurations, on the other hand, support early testing. Changes from other developers are seen when checked-in and are included during the next 'make'. The disadvantage is that this happens without any knowledge by the developer using the new file. A bad change by one developer may hinder the others on the project until the problem is fixed.

The second strategy put priority on early testing, but provides some protection between developers and is the model most frequently used at KCS. The drawback is that the individual developer can not get any version control between him checking out and checking in a file. Such fine-grained 'micro versions' would often be useful when rolling back a change that did not work. Support in this situation would also have the benefit of offering direct support for the individual developer. The only way to do this in ClearCase would be to introduce private branch-types for each developer and use a policy to do frequent merges to the project branch-type. For reasons mentioned above the management overhead when creating a new branch-type is too high to make this viable.

Summary – needed support:

- 'light-weight' branch-types
- fine-grained 'micro versions'

4.3 Merging and integration

According to the Ovum report [RBI95] ClearCase 'sets a new standard for industry' when it comes to the interface to the merge-conflict resolution. ClearCase thus offers a well developed support for merge. The command used is called ‘findmerge’ which means that when programs (or entire vobs) are merged ClearCase first *finds* which directories and files that have to be merged and can then, with the developers permission, really do the *merge* according to its default rules. After the ‘find’-phase ClearCase returns the files and directories that needs to be merged. The result is, however, just a list of names which makes it hard to get an overview. The form of the result seems more suited to be parsed by ClearCase again, launching the merge of the required files, rather than by a developer searching for inconsistencies. Considered that merge and integration often are made just before the release dead-line, a not very readable merge result often results in that the merge is done more or less in blindness, and as much automatic as possible. However, when conflicts are detected by the system and the developer is prompted to resolve it, the graphical merge tool is easy to use. Despite that the merge is line based and not so fine-grained, it is easy to understand and gives required support.

In section 2.1 a common work process, integrating modifications made at a project branch-type to the main branch-type, was illustrated by three steps. The process described is used at KCS, in the way that the CM-group has written conventions followed by the developers. Notable is that the configuration specification must be changed for every step in the process. This means that the specification, which is the single feature most prone to errors, must exist in three versions. A more directed support for this kind of common process should, most certain, reduce the number of errors made by the developers.

One should note that the process is iterative. After the tests have been made on the project branch-type and the next step should be to merge to ‘main’ the developer must make sure that no newer versions have been checked-in to ‘main’ since he did step (B). This is done by doing the find-phase of step (B) again, thus iterating the step (B) and test phases until the latest versions on ‘main’ have been tested and then quickly check in. Even with this iterative process we can not be totally sure that no files have been checked-in before we run step (C). If this happens, which is noticed by carefully reading the merge result, the convention on KCS is to revoke step (B) with an uncheck-out and do step (B) over again. Here a more active awareness of version checked-in had been of great help, see the awareness discussion in section 4.5.

Again, all steps can be made in ClearCase. However, when conventions must be used to force the developers to a certain behavior, we see that as a lack of support.

Summary – needed support:

- Support for integration of projects with a ‘main’ development line.
- More viewable merge results.

4.4 Support for delivery and consistent configurations

In ClearCase (like in e.g. RCS, SCCS, and CVS [Ced93, Wat]) keeping track of configurations are done by labels. I.e. for each file and directory a label, which is a special type of attribute, is attached to the version included in the configuration. It is then easy to ‘rebuild’ this configuration by just selecting the version of each file which is marked with the label. It might come as a surprise that in a configuration management system there is no support to record relations between such labeled configurations. For files each version is related to other versions of the file. A label, however, is just a text string (such as 'REL_2.0') and other means have to be used to keep track of existing configurations (i.e. Labels), why they are created and their relations.

KCS uses labels to label the version of all files included in a delivery, and also to mark configurations of systems and subsystems that have reached a certain level of maturity. In this way developers working on a subsystem easier can select correct versions of other subsystems without having detailed information about their development status.

Labeling files can be seen as taking a snapshot of the configuration. Between the snapshots, files are checked-in, both to branch-types, but also to main, changing the configuration continuously. It is therefore important to remember to label a configuration that later should be accessible. Normally this is done after a merge that has been tested, but before other changes has been made. It can otherwise be very difficult (although in principle possible) to 'find back' to the desired configuration.

When using labels the aspect of disc space consumption must be considered since labeling implies storing information about all files included in the configuration. With a product like TRIBON, including about 30,000 files, a labeling of the entire system (the main branch-type) will cost about 500 Kilobyte and take about 90 minutes to perform. Even if labeling is an easy operation to invoke the technique can not be used freely by all developers, but must again at KCS be controlled by the CM-group.

There is a need for individual developer to create stable versions of a system during development, for debugging, evaluating the impact of changes compared to earlier versions, backing out of mistakes involving several files etc. With labels these stepping stones must be pre-planned and created relatively often to be meaningful. Unfortunately the labeling technique is too heavy for routine use by individual developers. Again we notice that this is a situation where developers could directly benefit from the use of a CM system, but where well motivated restrictions makes it not possible.

Summary – needed support:

- a more powerful support for configurations than 'labels'

4.5 Support for group awareness

An important aspect of a system for people working together is its support for 'collaborative awareness', i.e. how and to what extent actions performed by others can be noticed by a user. In a CM system there are different demands from managers and developers and there are trade-offs between supporting isolation as opposed to awareness. The needs are also different in a setting where all users are working relatively close (say in the same building) or geographically distributed (as supported by ClearCase MultiSite). In the former case other mechanisms, such as informal meetings etc. can replace functionality in the CM system while in the latter case demands are higher.

Developer support needs

Development projects at KCS are often organized so the developer(s) can work in relative isolation from developers on other projects. Even so, there are situations where a developer needs to be aware of what other developers are doing. A frequent problem is to see if a file is already checked-out. This is supported in ClearCase with a tool creating graphs for individual files showing in which branch-types versions of the file has been created. Another tool creates a listing of all files currently checked-out. These tools thus contribute to awareness through giving an overview picture.

Using an optimistic update model, one developer checking in (or merging in) files on one branch-type might as a result change the version used by a developer working on another branch-type. In case there are incompatibilities, problems noted by surprise by the second developer, there is a need for support in making him aware of the change, at least a simple way for him to identify that a change has occurred. The ClearCase overview of versions of a file can be used to *verify* that a candidate file has indeed been changed, but there is no support for finding such candidates.

When a new branch-type is created other developers might need to update their views in order to include the new branch-type, although they are not primarily involved with it. Some kind of awareness of this kind of changes are needed. There is no support for this aspect in ClearCase and at KCS the conclusion is that also views might have to be managed by the central CM-group.

The experience at KCS is, however, that not much of the existing functionality in ClearCase supporting ‘awareness’ in the broad sense is in fact used, at least not by the regular developer. The exact reason for this is unclear, but it seems to be a combination of lack of match with the required support and that the provided commands are too awkward to use or have too long execution time. Instead most awareness is achieved by just ‘knowing’ what other developers are doing, i.e. through ‘social protocol’ rather than by system support.

Manager support needs

For the CM-group, managing the use of ClearCase, there is a need for overview of the system, in particular which branch-types that exists and how they are related to each other and to the views and labels. These aspects are poorly supported by ClearCase and at KCS some additional support has been developed by the CM-group. A tool generates html-documents which are stored on a local web-server. In these documents all currently existing views, branch types, and label types can easily be viewed by all developers. Viewing the current views, for example, also shows the corresponding configuration specification. This is to great help when creating new views and reduce the number of errors occurring in the specifications. The generation takes some time, though, and is therefore not made after every change which can result in some confusion when the list is not up to date.

Summary – needed support:

- collaborative awareness that a new version of a file has been created
- collaborative awareness for created branch-types
- relations between branch-types

5 Relevant functionality in the COOP/Orm environment

In this section we will outline how the ideas used in COOP/Orm can solve the problems identified at KCS. Since the problems are expressed in terms of functionality in ClearCase the mapping is not always direct to COOP/Orm functionality. In some cases the problems simply do not exist since they arise from using ClearCase itself. The presentation will follow the organization in section 4. A general observation is that while ClearCase seems more geared towards the needs of managers, COOP/Orm seems to support the needs of the developers as well. This is an important aspect when introducing a CM system, getting it accepted, understood and used.

Version selection problems (4.1) and need for consistent configurations (4.4).
The versioned configuration approach used in COOP/Orm seems to solve many of the problems identified in ClearCase. The explicit choice of versions replaces the need for 'view'-specifications and thus the problems they introduces. The need for repeated and error-prone changes to 'view' specifications is thus replaced with explicit selection in the version-graph. The explicit version graph in COOP/Orm makes the relations among revisions and variants explicit and solves the problem of relations among 'branch-types' in ClearCase. The need for versioned sub-systems is directly answered by the versioned configurations in COOP/Orm.

Support for short-term project (4.2).
The reasons not to allow developers to create new 'branch-types' in ClearCase are eliminated (or at least greatly reduced) by the explicit version-graph in COOP/Orm. It enables everyone to get an overview of the development history of a file or configuration. It should thus be less problematic to allow users to create short-lived branches. Furthermore, the fine-grained version control mechanism directly support the micro-versions for the benefit of developers.

Merging and integration (4.3)
The work process for merge followed at KCS, (see Figure 2), can be supported by choosing the order in which merge is performed in COOP/Orm. In addition the version graph will document that the process was followed. Support for the work process could also be improved, so for example the tool would insist on that the agreed process was followed. This is not an aspect in comparison with ClearCase, but is relevant when comparing with other systems such as Teamware [Team] and Continuus [Cont] that to some extent support work processes. The merge support in COOP/Orm is designed to give a good overview of the result and to let the user interactively explore different possibilities before settling on a particular choice. These mechanisms are currently extended to better support selection of consistent sets of related changes from the variants merged.

Support for group awareness (4.5).
COOP/Orm directly supports the demands for collaboration awareness identified at KCS, when creating a new version of a file as well as when creating a new variant.

COOP/Orm go beyond these demands with support for much finer grain of collaborative awareness. The need for this kind of interaction has not yet arisen at KCS, probably due to that project teams are typically put together with local members.

6 Suggested improvements in ClearCase

Some of the problematic situations identified in the previous section are such that there is in principle some functionality in ClearCase that seems to support the situation. However, it turns out in a number of cases that the functionality of ClearCase after all can not be used for the purpose due to some conflicting goal at KCS. In the following we will suggest some technical improvements of ClearCase which we believe is in the line of its overall design and will solve the problems.

6.1 Semantic checks for views

When evaluating a view it should be simple to report to the user any branch-type or label name that does not match an existing name. This rather simple improvement would considerably decrease the risk of using a bad view. This would be a significant step to continue to allow the users to manage their own view specifications.

6.2 Relations among branches

Although the very general form of branch-types, that can accept versions of files from any other branch-type, might be useful in some cases, the use at KCS shows that branch-types are in practice organized much more structured. Following a very common work process, ‘project’ branch-types are in fact used to house versions of files from a single ‘master’ branch-type. The versions of the files on the ‘project’ are eventually merged back to that same ‘master’ branch-type. This pattern can be repeated many times, with successive local ‘masters’ and several such ‘projects’ active in parallel at any given time.

Supporting such a restricted form of branch-types would have several benefits. It would directly support ‘light-weight’ branch-types allowing developers in a project to form local short-term projects in parallel with each other. ClearCase would also have the possibility to give an overview picture of existing such branch-types and how they relate to each other. Furthermore, since these branch-types are restricted, the corresponding view specifications can be much simpler, since they do not have to specify the rules ‘all the way up’, but inherit the rules of the ‘master’ branch-type. Together these mechanisms would make it possible for KCS to allow developers to create their own branch-types for local use, although branch-types for new projects still would be meaningful to coordinate.

The suggested ‘project’ branch-types are similar to the mechanisms in Teamware and would in the same way as for Teamware enable a minimal support for a work process. For the developer this would also mean that he could benefit from using version-control for his own work. The CM-group could with these mechanisms more follow the development rather than act as the central single point of CM-actions.

6.3 Versions of directories

Versions of directories in ClearCase only change when files are added/deleted/renamed in the directory. A more ambitious definition would be to create a new version of a directory when some of its files are updated. A version of a directory would uniquely specify the version of each included file as described in [Kat90] and in COOP/Orm [MAM93, MA96]. This much more advanced support for versioned directories would give support for versions of configurations. The possibility to show the difference between versions of the same directory/configuration provides a powerful change traceability mechanism and some of the hard questions when trying to get overview of the development would be easy to answer.

6.4 Awareness

ClearCase, and most other CM systems, are weak on providing 'collaborative awareness', i.e. information on what is happening with the systems due to actions by other developers. In many situations a developer wants to work alone, as if he, or his project members, were the only ones working with the system. In some critical situations, the needs are the opposite, one want to get an overview and immediate information about certain activities. An example we have mentioned earlier is when merging from a development branch-type. The actions of (1) updating the development branch-type with the latest versions on the 'master' branch-type, (2) testing, and (3) merging back could not be interleaved. If interleaving happens, the procedure has to be repeated. Awareness that some project has started such a process might be enough to make other groups hold back, and (on the other hand) noticing that some other group did indeed merge back would be a nicer way to realize that the procedure has to be repeated.

'Awareness' here indicates that the information we request is updated on-line as the changes occur. Aspects of awareness that we see meaningful to support is:

- For a 'master' branch-type, that some of its 'project' branch-types is in the process of merging.
- Monitoring creation of branch-types in general would be useful for the CM-group.
- For a branch-type, which files currently are changed or checked-out.

7 Future work / further studies

This paper reports on work in progress. Future work will be focused on evaluation of the suggestions put forward in this paper. Since the implementation of the suggestions in most cases would require added functionality in ClearCase, we will most likely try to illustrate the implications of the changes in the COOP/Orm environment. In this environment we plan to do small case studies with some of the source at KCS.

We also plan to work with a prototype implementation in ClearCase for evaluation of our suggestions: a semantic checker for views, support for structured branch-types, versioned directories and awareness through triggers and a client-server architecture. These experiments will have to use existing functionality in ClearCase (triggers, labels etc.) and conventions, so some of the cases might be too slow or cumbersome for professional use, but should be seen more as a proof of concept.

8 Conclusions

The use of ClearCase at KCS has at large been a positive experience. Its facilities has improved the situation for management of its large software system. Although ClearCase has proved very powerful and useful at KCS there are some situations where ClearCase gives less support than one might expect.

The main criticism of ClearCase is the lack of mechanisms for overview, error detection and structure. As a consequence its use needs more training, experience and overview than what most developers have. At KCS the result is that the CM support group has taken on more tasks and the use of ClearCase has been restricted for developers which are not allowed to perform crucial actions. In this way the overview over project and system development has been possible to maintain at least within the CM group. The restricted use also means that the use of ClearCase has been most beneficial for the development managers and less so for the individual developers.

Suggestions for improvements of ClearCase put forward in this paper includes: (1) Better error detection - to enable less experienced users to write specifications themselves. (2) Support for a new kind of 'branch-type' to support development projects - enabling more advanced support at integration and merge, better overview of parallel development, and allowing single developers to create local projects. (3) Advanced version control for directories to enable versioning of configurations. (4) ClearCase also has a general weakness in 'collaborative awareness'. Some information can currently be presented on user demand, while we in this paper suggest support for monitoring such information. We also ask for more extensive information for developers performing critical tasks to follow certain aspects of changes to the system performed by others.

We feel that with these improvements the deficiencies with ClearCase encountered during its use at KCS would be overcome and the overall usefulness of ClearCase would be much improved.

Acknowledgments

This study has been carried out in close cooperation with the CM-group and developers at KCS. First of all we want to thank KCS for opening their doors for us. We want in particular to thank Krister Erlansson, leader of the CM-group, for the time he has devoted to helping us in this study. NUTEK (The Swedish National Board for Industrial development) has supported this work through grant 93-3564.

References

[Ask94] Ulf Asklund. Identifying Conflicts During Structural Merge. In *Proceedings of the Nordic Workshop on Programming Environment Research*. Lund, Sweden. June 1-3, 1994.

[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.

[Clear] http://www.atria.com/products/clearcase.html

[Ced93] Per Cederqvist. Version Management with CVS. Available from Signum Support AB, Linköping, Sweden. 1993.

[Cont] http://www.continuus.com

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

[Kat90] Randy H. Katz. Toward a Unified Framework for Version Modelling 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.

[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

[MG96] Boris Magnusson and Rachid Guerraoui: Support for Collaborative Object-Oriented Development. In *Proceedings of ISCA International Conference on Parallel and Distributed Computing Systems*, Dijon, Sept. 25-27, 1996, pp 169-174.

[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.

[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.

[MMA97] Boris Magnusson, Sten Minör and Ulf Asklund: A Model for Semi-(a)Synchronous Collaborative Editing. In *Journal of Computer Supported Collaborative Work*. To appear.

[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.

[NFS] Sun Microsystems. The NFS Distributed File Service - A White Paper from SunSoft, stb 1252. 1994

[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.

[Team] TeamWare user's guides, Sun Microsystems, 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.

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