SOFTWARE CONFIGURATION MANAGEMENT IN SOFTWARE AND HYPERMEDIA ENGINEERING: A SURVEY

L. Bendix*, A. Dattolo**, F. Vitali***

** Department of Computer Science*
Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Øst – Denmark
bendix@cs.auc.dk

*** Dipartimento di Matematica ed Applicazioni*
Università di Napoli "Federico II"
via Cinthia, Monte Sant'Angelo, 80126 Napoli – Italy
antonina.dattolo@dma.unina.it

**** Department of Computer Science*
University of Bologna
Mura A. Zamboni, 7, 40127 Bologna – Italy
fabio@cs.unibo.it

Abstract

Software configuration management (SCM) is a very important feature in the software development area and in many authoring fields. The main purpose of this survey is to introduce readers to SCM in software and hypermedia engineering, presenting general concepts, principles and techniques and considering advantages and open issues. The new challenges proposed by World Wide Web (WWW), which can be seen both as a complex distributed hypermedia system and as a software development environment, are discussed.

SOFTWARE CONFIGURATION MANAGEMENT IN SOFTWARE AND HYPERMEDIA ENGINEERING: A SURVEY

Software configuration management (SCM) is a very important feature in the software development area and in many authoring fields. The main purpose of this survey is to introduce readers to SCM in software and hypermedia engineering, presenting general concepts, principles and techniques and considering advantages and open issues. The new challenges proposed by World Wide Web (WWW), which can be seen both as a complex distributed hypermedia system and as a software development environment, are discussed.

1. Introduction

In software development, the evolution process is characterised by continuous changes. Typically, a team of people produces, changes and exchanges common and individual software parts, working together towards a common goal. Often, the goal is not a single static object, but a dynamic collection of components designed to work with each other. Not all assemblies may result in a complete and consistent product, and the collection is often composed of a large number of components, with several persons at different sites maintaining and changing them; the entire development process often becomes a continuous history of changes, revisions and improvements. To keep all multi-version, multi-people activities under control, it is fundamental to introduce the concepts collectively called "software configuration management" (SCM).

SCM is the discipline of organising, controlling and managing the development and evolution of software systems.

The goal of using SCM is to ensure the integrity of a software artefact and to make its evolution more manageable; this objective is obtained by identifying the configuration of the software at discrete points in time and systematically controlling changes to the identified configuration for the purpose of maintaining software integrity, traceability and accountability throughout the software life cycle [1].

Naturally, the use of SCM implies an additional overhead in time, resources, and other aspects of the software lifecycle. However it is generally agreed that the consequences of not using SCM can lead to many problems and inefficiencies [2]. SCM is important for any design task, including software development, word processing, spreadsheet applications, hypermedia authoring, computer-aided design and manufacturing, databases and many other applications in which data is entered and revised frequently.

SCM has been particularly investigated in software engineering [3] and computer-aided design databases [4]. More recently, the advent of the WWW [5] and the possibility to use it as shared virtual workspace by a team of people have encouraged SCM research in hypermedia engineering.

The term "hypermedia engineering" [6] is quite recent; but it is defined using a similar definition as for software engineering [7], as the "employment of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of hypermedia applications".

The goal of this survey is to introduce readers to SCM in software engineering, showing general existing and proven techniques and explaining their main features. The second step is to deepen the discussion on SCM in hypermedia engineering.

The rationale of this choice is in the important roles played on the one hand by hypermedia systems in software engineering, and on the other hand, by configuration management concepts in hypermedia authoring. This is due in part to the explosion of WWW, that is at the same time the widest known hypermedia system, the largest and most complex software system, and the "meeting point" for distributed and collaborative workgroups. An additional reason to discuss SCM in hypermedia engineering is the strong interdependence that links software engineering and hypermedia engineering; in fact, several tools and concepts of software engineering are applied to the development, use and maintenance of hypermedia systems.

In section 2 we introduce SCM, providing a set of preliminary definitions and discussing traditional approaches to it. Section 3 presents general concepts, principles and techniques of SCM, considering advantages and open issues. Section 4 deepens the discussion on SCM in hypermedia engineering, presenting specific advantages and issues, and a brief history of existing hypermedia supporting SCM and highlighting the new challenges introduced in SCM by the advent of the World Wide Web. Finally, section 5 ends the paper with an account of the contributions of academic research and a look to the future perspectives.

2. Introduction to SCM

This section provides an introduction to SCM. In order to make it easier for the reader and avoid confusion, subsection 2.1 provides some preliminary definitions.

2.1 Some preliminary definitions

SCM has been investigated in many software development areas; in part due to this reason, there is a tremendous amount of overloading of terminology. Some attempts of unification have been proposed [4], [8][9], but, to assure there is no confusion in the reading of this paper, we provide the following definitions:

- **Configuration item.** A configuration item is each single software item that is individually identifiable as a logical entity. A configuration item can be atomic, like documents and modules, or composed, like libraries and systems.
- **Version**. A version is one specific instance of a configuration item. It provides a stable referencing mechanism for the management of configuration items. All versions but working versions are frozen, i.e. they cannot be changed without creating a new version.
- **Revision**. A revision is the kind of version that is a step in the evolution of a configuration item; it supersedes an earlier version and may contain arbitrary changes compared with its predecessor. Typically, it is created to fix bugs or to support permanent changes [10].
- **Variant**. Variants are alternative versions of a configuration item. They remain valid at a given instant in time and are typically created to handle environmental differences. A collection of variants of configuration items taken from specified revision levels defines the **version-set**.
- **Release**. A release is a version set that passed some defined quality assurance measures and is ready for a customer [11].

Versions, revisions, variants and releases are typically organised in a version graph, as in Figure 1, where the boxes represent the single configuration items.

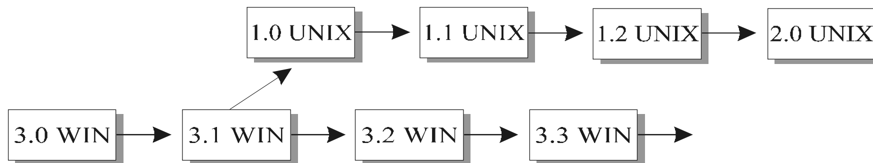

Figure 1. *Configuration items, versions, variants, revisions, releases.*

A software product is in version 3.0 for Windows (3.0 WIN). Horizontal arrows represent revisions (3.1 WIN – 3.2 WIN – 3.3 WIN – 3.4 WIN), while diagonal arrows represent variants (1.0 UNIX). A succession of revisions (1.1 UNIX – 1.2 UNIX) can produce a new release (2.0 UNIX).

We have defined SCM as the discipline for organising, controlling and managing the development and evolution of software systems. In this paper, we will use interchangeably the terms SCM and versioning, although the second term, versioning, will be principally used in the discussion of hypermedia engineering aspects.

2.2 Configuration management and version control

The timely and orderly development of quality software is both a challenging and difficult task. Software systems tend to grow ever bigger and the complexity of their development and maintenance becomes very hard to handle. Complex software systems such as the space-shuttle software or air-traffic control systems are amongst the most complex things ever constructed by man. They are the result of the cooperation of many people who have worked together as a large team, each person or group managing just a small part of

the complete system. We thus need methods and mechanisms to be able to divide a large system up into manageable parts and, even more important, to put those parts back together again to form a whole system.

In other situations, such as for telephone switchboard systems the management of the development of the system is further complicated by the fact that such systems are developed and maintained in many different versions and variants. The development of these different versions and variants must proceed in a controlled fashion and all versions and variants must be preserved for back-up purposes. Furthermore, since each part of a system may exist in several versions, we must have ways to control, record and query exactly which version of each part went into the final product.

These are some of the problems dealt with in the discipline, which was named "software engineering" at a NATO conference in 1968.

Configuration management and version control are two very important and fundamental activities in the software development process [12], and we believe that, together with results from fields like programming languages, development methodologies, and programming environments in general, configuration management leads to a distinct improvement in the general capability to produce high-quality software.

Some of the major problems in managing the development of large software systems are related to *the size of the system* (in lines of code), *the size of the project* (in number of people), and *the number of versions* of both the system and its components. Most modern programming languages contain a modularisation concept to help divide up a big system into smaller parts. This leads to the problem of having to assemble these parts to obtain a complete system, which, together with deriving object-code from source-code, is one of the problems dealt with by configuration management.

A multitude of different versions of the same item will come into existence during the lifetime of a project. This is due to the fact that bugs will be corrected and functionality will be changed or extended. We need to record information as to what has changed, why and by whom the change was made in order to be able to control the evolution of objects. Version control deals with all these problems.

Traditional approaches to configuration management and version control tend to view people as working in isolation and as such they give very little support for cooperation. A good support for cooperation will have impact on the way version control is performed and a proper integration between configuration management and version control becomes a mandatory requirement.

Today's systems tend to be long-lived and therefore most of the effort and money will go into actually maintaining the system. This implies that many changes - corrective as well as perfective - will be carried out during its lifetime, and it is therefore of the utmost importance to have good models, methods and tools for configuration management and version control such that good change control can be exercised.

Traditionally configuration management and version control have been seen as rather bureaucratic tasks (for instance in [13] and [14]), where clerical paperwork and standards are the main preoccupation. Other authors, like [15] and [16], looking at configuration management as a practical problem having a practical solution, have a strong tool perspective and focus much on the basic mechanisms and capabilities the programming environment must have in order to provide support for configuration management and version control [17].

3. Configuration management concepts and principles

SCM was born out of the 1968 NATO conference and specific military needs for much better control over software development efforts; it is an evolution of traditional configuration management. Configuration management was not something new, as it was practised in most other industries. However, to apply its principles to software development was – and still is to some extent – something new. For this reason, the military and its needs were the driving forces behind transferring and adapting configuration management to software development. Later on it was recognised that also non-military software development could benefit from exercising configuration management. This led to a shift in focus away from bureaucratic paperwork and control towards productivity and coordination.

According to [2], SCM is more complex than the traditional configuration management due to the malleability of the software: changes are easy to perform and occur more often that in traditional configuration management areas, software is simple to duplicate, which obliges SCM to manage multiple copies of a software component, some private, some public, each having its individual set of changes which may diverge in time, and, most importantly, the application of a change in a single component may induce hard-to-trace failures in other components due to non-evident dependencies among modules. On the other hand, SCM can be widely automated, since all the components are under computer control. This partly compensates for the added complexity.

The two single most important principles of SCM are the *immutability* of components and unique *identification* of components. Immutability of components is needed to ensure that no part of the development history – and thus traceability – of a component is lost. Once a component has been created and "published" it must remain unaltered and accessible forever. If need occurs to change the component, a copy must be taken, modified and subsequently added to the pool of immutable components. Once we apply this principle it is easy to see why we need unique identification too. If we cannot directly change a component, we will soon have several almost identical copies of it and thus we need to uniquely identify each of these copies.

The evolution of SCM is reflected in the structure of this section. First we explain the concepts and principles behind traditional SCM as defined by the military standards. Then we present how SCM is considered in a more pragmatic approach where military rigor is not needed.

3.1 Traditional configuration management

In the traditional perspective, which favours management needs, the objectives of performing SCM are to maintain product integrity, to obtain traceability, and to ensure accountability in software development.

The exposition in this subsection is based on approaches like [13] and [14] that show the connection from hardware configuration management to SCM. In particular, we highlight four operational aspects: configuration identification, control, status accounting and audit.

3.1.1 Configuration Identification

One of the fundamental principles of SCM is unique identification. The configuration identification activity covers both the identification of what things have to be covered and how they are named. This activity should ideally be carried out at the beginning of a project as a part of the planning phase.

We need to choose what should be put under control in the project. This includes far more than just the source code that will be produced. Things such as requirements, design, documentation, test cases are among the candidates to be put under control depending on the nature of the project. Each single entity that is individually identifiable is called a configuration item. Configuration items can be either atomic or composed; examples of the latter are configurations or sub-systems. Composed configuration items are, however, considered atomic entities at a higher level.

For each type of configuration item we have to plan how we want to structure configuration items belonging to that type. This structure will become part of identifying the configuration items. This is, however, not enough to uniquely identify items. For each type of configuration items we must decide on a naming scheme that allows us to identify it more precisely. Furthermore, we have to deal with the different versions of each single configuration item. Numbering schemes will have to be defined for versions, variants and releases.

3.1.2 Configuration Control

Configuration control addresses the old wisdom “if it ain’t broken, don’t fix it”. Changes to a system are made because it is broken in some way or some functionality is missing. This means that the central points for this activity are change requests and how they are handled from birth to implementation. Having this controlled and documented contributes towards the objectives of traceability and accountability.

Not all change requests should be implemented and not all changes are equally important. In order to decide what should be implemented and to prioritise changes we must form a committee. This is known as the configuration (or change) control board and consists of representatives of both supplier and customer. The

board makes its decisions based on information that may be provided by people outside of the board. Each change request must be documented, it has to be analysed and the impact of carrying it out has to be estimated. The configuration control board approves all changes before they are implemented and it also follows the implementation status of approved changes.

Another part of the configuration control activity has to do with the creation of a safe repository for all the configuration items of a project. The easy part of this is to find reliable storage; the more difficult part has to do with defining and enforcing access rules to the items. It must be defined who has access to what and how the access is performed. Usually access happens through a checkout operation that locks the item, in order to avoid that other people work on it; when the user has finished working on the item he releases the new version through a checkin operation.

3.1.3 Configuration Status Accounting

Configuration status accounting is the activity of keeping historical records of what has happened as far as Configuration Management is involved, and of using them in the CM process itself. Data must be collected and it must be reported. Usually the information is used by management and by the configuration control board.

Loads of data are created by the other activities during the development process and unless this data is organised in a way that we can analyse and act upon, it is of no use. The activity can be broken down into data capture, data processing, and data reporting. Reports must be written in such a way that they can be used. The number, type and form of reports needed will depend or the specific organisation's needs. Statistics is another type of data that is usually recorded as part of this configuration management activity.

Management use the status accounting reports to keep track of the current state of the system. Without this information it will not be able to know if the project is on track and what to do if it is not. Furthermore, management can assess the effects of the SCM process through the information available in the reports. This is a fact that is often overlooked by configuration managers who find a hard time justifying the resources spent on the configuration management process and tools. The change control board will need a different kind of reports from the status accounting activity. In order to perform their job they will need reports of proposed and approved changes and of problem reports. Furthermore, they will need information that enables them to follow the implementation status of approved changes such that problems can be discovered early and acted upon quickly.

3.1.4 Configuration Audit

The main purpose of this activity is to ensure the quality and maintain the integrity of both the product that is being built and the information that are being collected. Configuration audit is the activity of verifying and validating the fact that a proposed configuration is complete, accurate and consistent. It is also the activity where it is ensured that the recorded information is reliable and that the designated processes were followed

and that they work properly. It usually consists of functional configuration audits, physical configuration audits and configuration verification audits.

The functional configuration audit is conducted on the engineering prototype and its purpose is to assure that tests have been conducted to verify that each requirement has been met by the design. Even if formal tests cannot be carried out for some requirements, at least an error analysis must be performed to verify compliance. Physical configuration audit is started after the completion of the functional configuration audit. During physical configuration audit, software components (implemented modules) are verified by comparing them with the design units. Furthermore, revisions must be compared and verified. At the end of this activity an acceptance test is carried out to assure that the examined components are the same as those that passed the functional configuration audit. The configuration verification audit is conducted on both the entire configuration (system) and on the single components to ensure that integrity has been maintained after changes. It is important to verify that changes have been documented so that there is traceability between change requests and modified components. Furthermore, it is important to document and record the way in which components and configurations have been built. This is often done by recording bill of materials that state exactly how and from what a system – or subsystem – was constructed.

3.2 Practical configuration management

The early objectives of configuration management were to maintain product integrity and to ensure complete control and accountability of its development, which led to configuration management being practised as described above. This is, however, far too expensive an approach for small and medium size projects and in general for projects that do not need total control and accountability. Furthermore, traditional configuration management has very little help to offer the developer in his daily work, as its focus is on management needs. Emphasis has thus changed and more focus has been put on the coordination for team productivity [15], putting SCM into a wider perspective [18], and best configuration management practices [16] based on the traditional model. This has led to a different focus where the sub-division is: version control, coordination support, build management and configuration control [19].

3.2.1 Version Control

For many reasons it is convenient to keep, or to be able to recreate, a component. Version control is the activity that allows us to do this, and it has to address several problems. It has to minimise the storage space needed to keep all versions of a file. It has to impose a structure on how versions can develop from each other. And, finally, it has to keep track of all information about the different versions. This could be done manually, but using a tool these tasks are automated and are not prone to human errors.

Each single components of a given system will undergo many changes during the development and maintenance of that system. The set of changes that transform one version of a component into a new version

is called a *delta* and represents the difference between the two versions. In addition to the actual changes, we are also interested in keeping other useful information about the change such as who did it, when, for what reason. This information is called the *log entry* for a version. Together with the delta, the log entry constitutes one history step in the development history of a component.

The structure that a version control tool imposes on the development of history steps is called a *version graph* and is basically linear (Figure 2.a). One version follows the other and a new version is always created from the end of the line, as is the case for the tool SCCS [20]. For simple development needs this model is sufficient even though limited. However, it cannot support maintenance (i.e., further development of) older versions, it does not handle parallel development (for instance, additional development and maintenance going on in parallel) and thus variants of the same component cannot be represented in this model. To solve that problem most tools allow branches to be created from older versions and thus support the tree model (Figure 2.b) that was introduced in RCS [21]. In this model several branches can exist in parallel to reflect either maintenance of older versions, parallel work or variants. Other models support acyclic graphs (Figure 2.c); in this way, a version may have two or more predecessors, for example, in order to express that a bug fix in an old version is merged with the currently developed version.

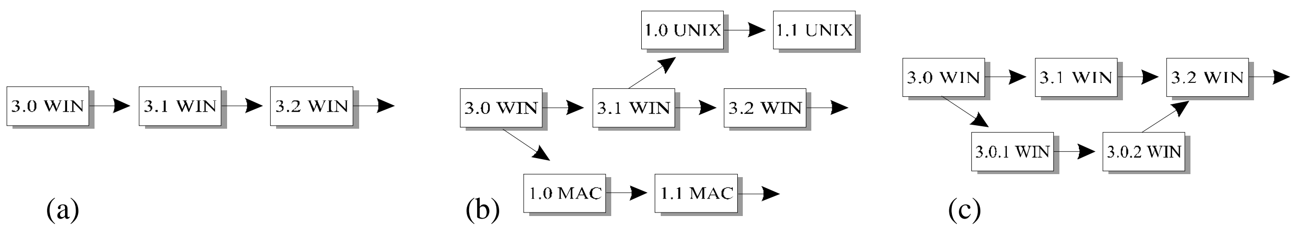

Figure 2. *Version graph structures: linear (a), tree (b) and acyclic graph(c) models.*

The degree to which it is possible to represent information about versions varies between tools. Most tools, however, have the possibility to save textual information about a version, which means that any information could be represented that way. The drawback of having information represented as text is that it becomes difficult to search for information. Most tools have a number of attributes connected with each version and these attributes can be used whenever we want to retrieve a version from the repository.

3.2.2 Coordination Support

Today all but the smallest projects are carried out as group efforts. Many people work together on the same product towards a common goal. We need to coordinate the effort of these people such that they do not interfere with each other's work. On the other hand we should also allow them to be as productive as possible.

Many problems can happen when people share things and want to change them, as pointed out by [15]. The core of the problem is that we really do not want people to share things that are being changed, even though their parts go towards the common system. If something that is under my control is changed by someone

else, I might not be aware of it and I might be misled to believe that I introduced an error, when in reality it was introduced by their change. So it is important to work isolated from the changes that other people make. Sometimes people may share documents and may want to change the shared document. If that is not coordinated properly, the change of one authors may be lost, as it gets overwritten by the other changes. To have identical copies of shared documents is not a viable solution, as it will soon become even more difficult to keep the copies identical.

In order to solve these problems we can try to keep the components as small as possible to make sure that there is no need to share components. This is often not possible and we have to allow some degree of sharing of components. If components are shared we will have to deal with the fact that there might be the need to carry out changes in parallel. Two different strategies are used here: one is to forbid it, the other to support it. When it is forbidden changes have to be sequentialised. This is done by locking components before they can be changed, the lock making sure that only one person at a time can change the component. Tools that allow components to be changed in parallel simply create a branch to reflect that two parallel changes are going on, as is the case using CVS [22]. Most tools are also able to merge the two branches together back into one version to get the composite of the two changes.

The problem of other people's changes having an impact on the system without us being aware of it is handled differently. What we would like to obtain are periods where we work in isolation from other changes, and periods where we want to integrate our own changes with those of other people. We can use the workspace for that purpose. The central repository handles stable versions of the configuration items. In the workspace we copy (check-out) the items we intend to work on, and also a copy of all components of the system. When a change is checked into the repository by someone else, the copy in our workspace will remain the same. When we check in the contents of our own workspace, we will check in only those components that we have actually changed. Furthermore, as most tools allow parallel work, there is the possibility that someone else has made a change to a component that we have changed too and the two changes have to be merged. If this is the case the later person to check in has the responsibility to make sure that the merge has been carried out correctly and that all conflicts are resolved.

3.2.3 Build Management

Build management handles the problems created by following the good programming advice to divide large programs into modules [23]. These modules have to be put back together and compiled in order to create a running system. The number of modules or components in a system is increased by the fact that we might want to keep them as small as possible to avoid sharing conflicts.

To build a system from its components we need both a description of the structure of the system and information about how to derive object code from the source code modules. The description is called a *system model* and is used to instantiate specific systems. In early tools like make [24] both the system model and the information about how to derive object code and link it together was given in one single resource, the

makefile. This automates the build process and avoids errors from human intervention. Furthermore, make implements a minimal build as it only recompiles source modules that are out-of-date with respect to existing object code. This can lead to huge savings in compile time for big systems, if only a small part of the system has been changed.

Further improvements have been made to build tools to remove the limitations of the original. The most severe problem with the original make is that it keeps no data of its own. To decide whether an object module is up-to-date or not it uses timestamps from the file system. This can be very unsafe especially in distributed environments where there is no global clock. Furthermore, make does not remember the way a given object module was created. This means that source modules do not get recompiled if such things as compilation options are changed or another version of the compiler is used, and this may create several problems. Additionally, the successors of make take advantage of the fact that today's computers are networked and support for compilation in heterogeneous environments is common as is the support for parallel compilation.

3.2.4 Configuration Control

The build management we just described has no notion of versions of modules. Each module that has to go into the system, described by the system model, is assumed to exist in only one version and the task managed by them is only to compile and link these modules as efficiently as possible. This works quite well for projects without version control – and in the case of working in a workspace that has all the needed modules locally. However, when we take a more global look some weaknesses of this approach become evident.

If we add versions to the modules this has to be reflected in the system model. For each node in our dependency graph we now have a version group instead of a single file. This is much the same we have when we look at the repository and want to check some (or all) modules out into our workspace. This means that the build tool must be extended with the ability to select from the repository. Unfortunately, not all queries to the repository are unambiguous. In the case of a human being checking something out, this is not a problem as he can refine the query to become unambiguous. However, a tool does not have that capacity. This means that we have to be very careful about how we write our selections. Most tools have opted for the rather unsafe solution to automatically (without user intervention) solve all ambiguities by choosing the latest version. However, once the selection has been made we are back to normal build management.

In the absence of versions (in our workspace) we can still run into troubles. If the structure of the system changes, this will have to be reflected in the system model. This implies that we will have to create a new system model for each change in the structure. So we will have to perform version control not just on the single modules, but also on the system model. Even worse we will have to make sure that the system model that is used to construct the system is also the same that was used to create our workspace. No existing tool does that automatically, so it all has to be done manually – exercising extreme care. Fortunately most

systems do not change structure very often, but if rapid prototyping is used as development model this issue may become prominent, and currently there is little help from existing tools.

4 Versioning Hypermedia and the Web

In the hypermedia field, the problems connected to configuration management have been frequently examined and discussed [25]. Of course, most of the problems that apply to other authoring or software development environments also apply to hypermedia. Yet, there are peculiarities in versioning inter-linked content, that at the same time pose some new problems, and propose an elegant solution to an important issue, referential integrity of links [26], that is specific of hypermedia. This may in part be the reason traditional approaches to versioning have been customized to the requirements specific of hypermedia.

Several important systems throughout the long history of hypermedia have discussed, implemented, or even relied on versioning functionalities, from Nelson's Xanadu [27] to the current IETF standardization effort WebDAV [28], ranging through well-known systems such as SEPIA ([29] and [30]), Hyperform [31], and Microcosm [32]. One most important hypermedia system, the World Wide Web, has further exacerbated the complexity, the vastity and the diversity of the issues involved. In this section we will first introduce the basic issues of versioning hypermedia, then a brief roundup of the systems and proposals that have made the history of this field, and then we will concentrate in considering the issues of versioning specifically to the World Wide Web, which can be seen both as an overly complex, anarchistic distributed hypertext system, and as the environment of choice for the sexiest software applications lately.

4.1 Versioning hypermedia

The versioning is considered as one of the relevant issues in modelling and building of hypermedia systems and its importance has been emphasised in [33] as regards its relationships with software engineering.

In situations where the hypermedia authoring is managed and controlled both within and outside of a workflow, the accounting and verification of the authoring activities provide important motivations for versioning; just like in the software engineering process, it is useful to compare the evolving states of the same resource in order to:

- determine the changes,
- evaluate the progress of the development;
- preserve former states and save them from destruction;
- reuse intermediate states;
- preserve the historical perspective of work done;
- maintain the consistency of the distributed resources [10];

- provide effective solutions to multi-user concurrency control [34].

Of course, versioning will only be accepted by the user if the effort spent on configuration management is out weighed by the benefits. This imposes that the versioning must be carried out automatically, according to the session-based schema, instead of the user-decided approach that is required to let the user free to adopt the choices he prefers [29].

4.1.1 Advantages

The advantages deriving from applying version management benefit many aspects of the authoring process:

- *Exploratory authoring* is helped by versioning, since keeping a good and reliable baseline version makes authors more confident in trying out new development paths with their documents, even if these paths eventually turn out to be unviable or unacceptable.
- *Distributed and collaborative authoring* may profit the most from version support: verification and evaluation of others' contribution is eased by the possibility to compare their work with previous baseline versions of the resource; furthermore, if the chain of subsequent versions is allowed to fork into independent branches, then multiple authors can work on the same resource at the same time with no risk of overwriting.
- *Long transactions*. It has been noted (for instance see [34]) that in many creative authoring situations (including hypermedia) long transactions are crucial, because operations in authoring environments are often long-lived. In these situations of collaborative authoring efforts, especially if distributed geographically, it becomes fruitful to use *locking*, that prevents other users but the first to access and modify a resource, and *branching*, that forks the single flow of versions into two or more independent sequences, possibly later merged back together. Branching versions could even encourage emergent unplanned collaboration [35], whereby even simple readers of a published resource, if providing interesting suggestions, modifications, or additional links, could be accepted and published side-by-side with the main version of the resource.
- *Referential integrity of links.* Versioning constitutes an additional advantage for hypermedia, providing an easy solution to the old problem referred to as referential integrity of links [26], which can be shortly described as the problem of automatically updating references to positions in documents that have changed In many situations, it is useful to link not just whole documents, but specific locations within them (the so-called point-to-point links). There are very good reasons to store links outside of the documents they connect, rather than embedded inside them *à la* HTML. For instance, external links allow linking into resources that we have no write access to, that are stored in a read-only medium, or that have been heavily linked to by other users for other purposes we are not interested in. Furthermore, external links allow private links, specialised trails, and multiple independent link sets to be created for the same document set. It is interesting to note that the W3C is proposing with XLink [36] a way to crate external links on Web documents. Whenever a linked document changes, external links risk to point to

the wrong place because of the changes themselves: whatever method is used for referring to a specific location of a document (e.g., counting, searching or naming), there exists a type of change that messes up the reference. To summarise, the referential integrity of links is at stake whenever the modification of a document does not update at the same time all the existing references to it, and especially when these references point to a specific location of the document. When no update is performed, we end up with dangling links, i.e., links that do not point to the correct destination. Automatically fixing dangling references can be performed either by applying a *"best bet" heuristics*, whereby the current link end-point is determined by finding the most similar content to the old end-point, or by *position tracking*, that is, by following change after change the evolution of the references, thus determining their current positions. Of these solutions, only position tracking guarantees a correct solution in all cases. Position tracking, on the other hand, requires fine-grained versioning, i.e., it requires the changes to a document to be correctly and orderly recorded, since it is by comparing them in a sufficiently fine-grained detail that it is possible, following version after version, to track the evolution of the relevant positions.

4.1.2 Structural and Cognitive Issues

The adoption of a versioning mechanism within a hypermedia system raises some important structural and cognitive issues:

- *Version models*. In [37] two basic version models were identified:
  - *state-based versioning* maintain the version of an individual resource; does not support the tracking of a set of changes involving several components of a hypertext network.
  - *task-based versioning* focus on tracking versions of complex systems as a whole; it provides system support for maintaining the relationships between versions of resources that have been changed in a coordinated manner during the performance of a task. Holistic, task-based approaches to versioning are especially sensible in hypermedia, given the complex, multi-dimensional aspects of modifying a complex hypermedia network.

  These concepts are similar to those of state-based and change-based versioning as known in software engineering [38].
- *Immutability of frozen versions*. As discussed in [39], it represents an important structural problem. Intrinsic to any version model is the fact that older versions of a resource are frozen, that is, they cannot be changed without creating a new version of the resource. Yet it may be useful to allow frozen versions to have new links (for instance, annotations or comments) coming from and going to them without necessarily creating a new version of the resource. At the same time, some links are substantially part of the resource itself, and thus their modification should definitely require the creation of a new version of the whole resource. Depending on their meaning and their role, therefore, the creation of some new links may or may not imply the creation of a new version of the resource. External link sets may seem a solution to this problem: when creating a new version of a resource, the author would also specify the set

of substantial links (the ones that if modified would create a new version of the resource), while all other links would be considered as annotation links (and would not require a new version of the resource if changed or incremented).

4.2 Hypermedia systems supporting versioning

Version management is intrinsic and fundamental to the inner workings of the Xanadu system [27], one of the oldest proposals for hypermedia system. Xanadu proposes a peculiar way to organise the data, called the Xanalogical storage [40], where the documents (the minimal structure of the system) either actually contain their content (native bytes), or refer to it by inclusion from other documents (included bytes). In Xanadu, versioning is at the same time an immediate functionality of the system (a new version of a document is a new document that includes all the parts of a document that were present also in the previous version, and that has as native bytes all the new data) and a fundamental requirement for it to work: in Xanadu inclusions refer to their end-points by offsets, so that any change to the content of a document would corrupt the very structure of the inclusions unless exact tracking of the documents' evolution is activated through versioning. Later in time, both RHYTHM [41] and Palimpsest [42] proposed solutions similar to the Xanalogical storage, heavily relying on versioning for the management of correct inclusions.

Open hypermedia systems are largely used to address the complexity and heterogeneity of large-scale software development [43]. A body of research has been developed over the past two decades that has explored the effectiveness of this approach in supporting software engineering tasks; hypermedia was first used to support the construction of software systems in Neptune [44]. Neptune was built on top of the Hypertext Abstract Machine (HAM) [45] and featured integrated support for hypermedia within custom-built CAD applications. The elaboration of these first approaches has conducted towards interesting research that has been performed on:

- integrating hypermedia functionality [46] into all the tools of an integrated development environment (IDE) [47];
- examining how hypermedia services can influence the act of programming itself [48];
- developing and creating hypermedia-enabled programming environments [49], [50];
- using hypermedia in an attempt to raise the level of abstraction at which software engineers perform their software development tasks [51].

After the experience of the PIE system [52] and Halasz's powerful keynote address to the Hypertext '87 Conference [33], where versioning was mentioned as one of the main open issues in the hypermedia field, many researchers set out to study the subject: CoVer ([29] and [30]) is a contextual version server that can provide both state-based and task-based version models for the SEPIA hypermedia authoring system; within HB3 [53] and Hyperform [31] researchers concentrated instead on abstracting the concept of version management from the actual hypertext systems they were going to provide such service for. An analytical

approach to version management is presented in [54], where structural and cognitive issues are managed in a Petri net based hypermedia model.

In 1994 and 1995 two versioning workshops ([55] and [56]) helped to further shape the field, examining aspects such as link selection, conceptual overhead in version freezing, and support for collaboration in distributed hypermedia. Among the hypertext systems that discussed the implementation of some kind of version support we will also include Microcosm [32] and Chimera [57].

A first attempt to manage versioning by means of a fully distributed and cooperative approach based on autonomous actors is proposed in [58] and successively extended with HyDe [59]. The formal model at the basis of HyDe is constituted by populations of independent and autonomous actors that work in a distributed and concurrent universe in order to reach the common goal of configuration management. It represents a first step in the definition of a reference model for the World Wide Web that, given its importance and diffusion, is discussed more in detail in the next section.

4.3- Versioning the World Wide Web

Even considering the WWW just as a distributed hypermedia systems, then the issues discussed previously also apply: the WWW is being provided with an external linking mechanism called XLink [36], which will provide sophisticated linking mechanism to XML [60] documents, and that will most prominently face the problem of referential integrity of its links. Furthermore, inclusion mechanisms for whole documents and parts thereof are already available even in HTML 4.0, and probably the lack of any position tracking mechanism for the end-points of the inclusions will hinder their use. An early proposal to deal with these issues was VTML (Versioned Text Markup language) [61], a markup language to express change operations for WWW documents, in particular HTML. It should be noted that already HTML 4.0 [62] includes two tags, INS and DEL, that are meant to express changes from previous versions of the same document (e.g., in legal texts); unfortunately these tags, being part of the markup language, cannot express changes in the markup itself (e.g., that two paragraphs have been joined, that a link destination has been changed, etc.) or changes that disrupt the correct nesting of the markup.

But the WWW is much more than a distributed hypermedia system: it is more and more the target platform of choice for software development, content provision, e-commerce, etc. As noted in [61], creating successful web sites requires the knowledge of a wide array of diverse and non-conventional tools, technologies and languages, by a large number of different authors with different skills, background, professional skills, few of which will probably have a background in change management and design. Other problems in the development of web sites can be found in the pace of development for Internet sites, which hinders the adoption of mature, process-driven development techniques, the number of slightly but not quite compatible client environments to build for, that forces the management of a very large number of variants of the same solutions, and the problems connected with the outsourcing and parcelisation of the development

and deployment, which, although in general a positive trend, may also add delays, misunderstandings, and incoherence in the development of web solutions. Even if no software applications are involved, but just content production, the potentials for corporate embarrassments or legal liabilities do exist in the production of any web site that does not provide a formalised process for change management and testing.

WebDAV ([63] and [64]) is an IETF working group devoted to extend the HTTP protocol in order to provide a general, standardised framework for distributed authoring and versioning management for Web sites. The current specifications deal with distributed authoring, defining new methods, new headers and response codes for HTTP, a workable formalisation of the existing PUT method, as well as the definition of "properties", meta-information parcels associated to editable resources. Two further working groups have been established within the main WebDAV activities, DASL (DAV Searching & Locating working group) [65] to provide querying mechanism for WebDAV-enabled sites, and Delta-V [66] to further extend the protocol for version management of web resources.

The WebDAV protocol provides concurrency control for write access to the same resources, allowing locks to be set and unset on shared resources. It provides a way to add, modify, delete and access arbitrary metadata of Web resources, so that the mere content of resources is enriched by an open set of properties, name-values pairs that can be freely associated with Web resources. It provides support for content independent links and relationships between resources. It provides support for server-side namespace manipulation, allowing authors to move, rename and copy web resources on a server. It provides support for collections, containers of resources allowing grouping, navigation, query, etc. Finally, the WebDAV protocol has set as a fundamental requirement that of providing the aforementioned services to all Web resources, regardless of their data type. This helps to make the HTTP protocol a truly interoperable distributed environment for collaboration and authoring.

Within this framework, the Delta-V group is providing further extensions in terms of methods, properties, response codes and headers, so as to provide version management for Web resources. The WebDAV versioning model includes the management of multiple versions of the same resources, independent access to each revision, parallel access and modification of the same versioned resources (branching) and the subsequent reunification of branches (merge), configuration management and collection versioning, under the same set of global requirements that WebDAV already has.

WebDAV does not provide direct support for the design and activation of change management processes, which are seen as fundamental to bring some order in the current chaos of web development. Furthermore, it does not provide direct support for small-granularity change tracking mechanisms, necessary for the automatic maintenance of the referential integrity of links. Unfortunately, these mechanisms are inherently content-type dependent and therefore outside of the scope of the WebDAV protocol.

Nonetheless, this protocol represents a fundamental step forward for the evolution of the Web towards a real distributed collaborative environment, and a necessary platform for these missing features and practices to be built.

5. Configuration management research and concluding remarks

Over the past twenty years, SCM has been largely investigated and it has become an indispensable support for the development and authoring of software products. The majority of research related to SCM is found in software engineering, but many issues encountered in this area are relevant to the hypermedia engineering, including version control, storage management or object identification [67]. The field of hypermedia has dealt with versioning issues for a long time, since Xanadu considered it a fundamental mechanism for its inner workings. The main difference in these two research areas is in the hypermedia management of explicit relationships among the resources being managed.

Versioning hypermedia presents a few new problems because of the management of ad hoc relationships among versioned resources. On the other hand, besides many more advantages, versioning provides an easy and safe solution to the well-known problem of referential integrity of links.

On the basis of this and similar observations on the reciprocal support provided by software engineering to hypermedia engineering and viceversa, and principally in consideration of the new challenges proposed by the WWW, this paper has been focused on SCM not only in software engineering but also in hypermedia engineering.

This survey has presented general concepts, principles and techniques, considering advantages and open issues; in this concluding section, we want to look into the research achievements of configuration management, both past and present. Furthermore we will point to what we anticipate to be the future problems of configuration management that have to be solved by new research. Academics and academic research has been and still is rather absent from what is happening in the configuration management world. The military started the field of SCM because of their need to control their development of software. Large companies and tool vendors followed after as the drivers of further progress. Academic research has, though, made some substantial contributions on the technical level, which has brought significant improvements in the tools and models used for configuration management today. However, the practise of configuration management (its activities and process) also today continues to be dominated by companies, practitioners and tool vendors. The problem with this approach is that results are rarely widely published and improvements thus spread very slowly.

5.1 Past

The following are some of the achievements from past academic research. The most basic requirement to a configuration management system is that it must provide an efficient and secure storage. Today's SCM tools provide secure storage, even though some tools have occasional corruptions caused by the rather complex structures that have to be maintained. The efficient use of space to save multiple versions is assured by the use of delta techniques [20], [21]; these techniques have even been refined to be applicable to binary formats too [68]. Workspaces have been provided to ease moving back and forth files by allowing to group them in hierarchically structured projects [22]. Furthermore, mechanisms exist to coordinate groups of people by letting them work in parallel and in isolation from each other, and only synchronise and automatically merge their work when they are ready [22]. For the production of systems, automation is provided by tools like make [24], distributed and parallel builds on heterogeneous platforms were provided by [69] and [70], and before that reproducibility and traceability from a system to its components was supported by bill of materials in [71]. Flexible and powerful system models have been proposed by both [71] and [72]. Support for specifying process aspects of the configuration management activities was investigated in [73].

5.2 Present

Several problems are presently being worked on in the academic world and are giving insight that will mature into results to be adopted in the next few years. In today's reality for software development, teams that work on the same system are often distributed around the world. There are even companies that have "around-the-clock" software development, where one team picks up the development work when another goes home after its day of work. This leads to high demands for a SCM tool's capability to synchronise people that are located far from each other and that might have only slow modem lines to transfer files and to check in their work [74]. The same situation occurs when people want to work from home. Many companies are discovering the benefits of performing configuration management, but are not quite sure how to do it. Work is being done on collecting experience and best practices from companies that have already implemented configuration management [75]. Likewise, teaching of configuration management concepts and principles has been neglected at many universities with the result that people working in industry have little or no knowledge of what SCM is or how to perform it. There is the need for a huge education effort here [76]. The fact that a configuration management system collects information about the components being developed has led to research into whether software configuration management tools and product data management (PDM) tools can and should be integrated [77].

5.3 Future

There are still enough problems around to keep configuration management research alive for many years to come. Just as the connections between SCM and PDM are being researched, also the connections between software configuration management (system models) and software architecture should be researched. Principally thanks to the support of the WWW, it is becoming more frequent to see that people are working in integrated teams that are physically distributed. Such teams need ways to create awareness about what is going on and how the project is progressing. Furthermore, we need ways to be able to synchronise and merge work that is not based on lines of text. Component-based software development has been proposed as a way to improve software productivity. If components are to be substitutable also at runtime, we need SCM tools that are capable of handling dynamic configurations. Many companies are discovering that they are producing many different products that are alike. It is tempting to try to reuse as much of the commonality between these products as possible go gain in productivity and time to market. This, however, means that they have to handle many variants of the same component and to keep track of which variant of a component went into what product. Variant handling has already been researched, but no satisfactory solution has been provided yet. Especially not a solution that covers all aspects from variation in the small (like running on different operating systems) to variation in the big (like different graphical user interfaces). More research should also be done to document the effects of configuration management. So far only one experiment on the return of investment of introducing configuration management has been reported [78].

Finally, there is the eternal problem of how to get configuration management functionality properly and transparently integrated with the rest of the tools in an environment, such that it is "invisible" for the user under "normal" circumstances. In DSEE [71] that was actually the case, as all calls to reads and writes in the operating system were intercepted by the configuration management functionality. Unfortunately this approached seems to have been abandoned by tool vendors.

6. References

[1] A. Leon, *A Guide to Software Configuration Management*, Artech House Publishers, April 2000.

[2] A. Zeller, *Configuration Management with Versions Sets*, PhD thesis, Technical University of Braunschweig, April 1997. http://www.cs.tu-bs.de/softech/papers/zeller-phd/

[3] S. Dart, Concepts in Configuration Management*, Proc. of the 3rd International Workshop on Software Configuration Management*, Trondheim, Norway, June 12-14, pp.1-18, 1991.

[4] R. H. Katz, Toward a Unified Framework for Version Modeling in Engineering Databases, *ACM Computing Surveys*, 22(4):375-408, 1990.

[5] T. Berners-Lee, R. Cailiau, A. Luotonen, H. F. Nielsen, A. Secret, The World Wide Web, *Communications of the ACM*, 37:76-82, 1994.

[6] D. Lowe, W., Hall, *Hypermedia and the Web. An Engineering Approach*, John Wiley & Sons Ltd, 1998.

[7] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, Spring Edition, 1991.

[8] W. F. Tichy, Tools for Software Configuration Management, *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, January 27-29, 1988, Teubner Verlag, 1988.

[9] K. Meiser, Software Configuration Management Terminology, . http://www.stsc.hill.af.mil/crosstalk/1995/jan/terms.asp

[10] B. Magnusson, Fine-Grained Revision Control for Collaborative Software Development, *SIGSOFT'93*, pp.33-41, 1993.

[11] S. A. MacKay, The State of the Art in Concurrent, Distributed Configuration Management, Software Configuration Management: *Selected Papers SCM-4 and SCM-5*, Seattle, WA, April, J. Estublier ed., *LNCS (1005),* Springer-Verlag, pp. 180-194, 1995.

[12] M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber, Capability Maturity Model, Version 1.1, *IEEE Software*, July 1993.

[13] E. H. Bersoff, V. D. Henderson, S. G. Siegel*, Software Configuration Management: An Investment in Product Integrity*, Prentice-Hall, Inc., 1980.

[14] F. J. Buckley, Implementing Configuration Management: Hardware, Software, and Firmware, *IEEE Computer* Society Press, 1993.

[15] W. A. Babich, *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley Publishing Company, 1986.

[16] D. D. Lyon, *Practical CM – Best Configuration Management Practices*, Butterworth-Heinemann, 2000.

[17] V. Ambriola, L.Bendix, P. Ciancarini, The Evolution of Configuration Management and Version Control, *Software Engineering Journal*, 5(6), November 1990. (reprinted in "Software Engineering: A European Perspective", Richard H. Thayer, Andrew D. McGettrick (eds), IEEE Computer Press, Los Alamitos, USA, pp. 389-401, June 1993)

[18] M. Kelly, *Configuration Management – The Changing Image*, McGraw-Hill Book Company Europe, 1996.

[19] L. Bendix, Fundamental Tasks in Software Development Environments, *INFORMATICA - An International Journal of Computing and Informatics*, 19(3), September 1995.

[20] M. J. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering*, SE-1(4), December 1975.

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

[22] B. Berliner, CVS II: Parallelizing Software Development, *Proc. of USENIX Winter 1990*, Washington D.C.

[23] D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, 15(12), December 1972.

[24] S. I. Feldman, Make - A Program for Maintaining Computer Programs, *Software - Practice and Experience*, Vol. 9, April 1979.

[25] F. Vitali, Versioning Hypermedia, *ACM Computing Surveys*, 1999.

[26] H. C. Davis, Hypertext Link Integrity, *ACM Computing Surveys*, 1999.

[27] T. H. Nelson, *Literary Machines*, Edition 87.1, Sausalito Press, 1987.

[28] J. Slein, F. Vitali., E. J. Jr. Whitehead, D. Durand, Requirements for Distributed Authoring and Versioning on the World Wide Web, *ACM StandardView* 5(1):17-24, 1997. Also published as RFC 2291, February 1998, IETF, http://www.ics.uci.edu/pub/ietf/webdav/requirements/rfc2291.txt.

[29] A. Haake, CoVer: a Contextual Version Server for Hypertext Applications, in *ECHT '92 Conference* (Milan, I), ACM Press, New York, pp. 43-52, 1992.

[30] A. Haake, Under CoVer: the Implementation of a Contextual Version Server for Hypertext Applications, in *ECHT '94 Proceedings* (Edinburgh, UK), ACM Press, New York, pp. 81-93, 1994.

[31] U. K. Wiil, J. J. Leggett, Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems, Proc of *ECHT '92 Conference* (Milan, I), ACM Press, New York, pp. 251-261, 1992.

[32] M. Melly, W. Hall, Version Control in Microcosm, *Proc. of ECSCW '95 Workshop on the Role of Version Control in CSCW Applications*, Stockholm, Sweden*, 1995.*

[33] F. G. Halasz, Reflections on Notecards: Seven issues for the next generation of hypermedia systems, *Communications of the ACM*, 31(7):836-852, 1988.

[34] U. K. Wiil, and J. J. Leggett. Concurrency Control in Collaborative Hypertext Systems, *Hypertext '93 Proceedings* (Seattle, WA), ACM Press, New York, pp. 14-24, 1993.

[35] C. Maioli, S. Sola, F. Vitali, The Support for Emergence of Collaboration in a Hypertext Document System, *CSCW'94 Workshop on Collaborative Hypermedia Systems* (Chapel Hill, NC), GMD Studien n. 239, 1994.

[36] S. De Rose, D. Orchard, B. Trafford, *XML Linking Language (XLink)*, W3C Working Draft, 1999, http://www.w3.org/TR/WD-xlink

[37] A. Haake, D. Hicks, VerSE: Towards Hypertext Versioning Styles, *Hypertext '96 Proceedings* (Washington DC). ACM Press, New York, pp. 224-234, 1996.

[38] R. Conradi, B. Westfechtel, Version Models in Software Configuration Management, *ACM Computing Surveys*, 30(2), June, pp. 232-282, 1998.

[39] K. Østerbye, Structural and Cognitive Problems in Providing Version Control for Hypertext, *Proc. of ECHT '92 Conference* (Milan, I), ACM Press, New York, pp. 33-42, 1992.

[40] T. H. Nelson, Xanalogical Media: Needed Now More Than Ever, *ACM Computing Surveys*, 1999.

[41] C. Maioli, S. Sola, F. Vitali, Wide-Area Distribution Issues In Hypertext Systems, *SIGDOC '93 Proceedings* (Kitchener, Canada), ACM Press, pp. 185-197, 1993.

[42] D. Durand, Cooperative Editing without Synchronization, *Proc. of Hypertext '93 Workshop on Hyperbase Systems* (Seattle, WA), Technical Report n. TAMU-HRL 93-009, Hypertext Research Lab, Texas A&M University, College Station TX, 1993.

[43] K. A. Anderson, Supporting Software Engineering with Open Hypermedia, *ACM Computing Surveys*, **31**(4), December 1999.

[44] N. M. Delisle, M. D. Schwartz, Neptune: A Hypertext System for CAD Applications*, Proc. of ACM SIGMOD'86*, Washington DC, May, pp. 134-142, 1986.

[45] B. Campbell, J. M. Goodman, HAM: A General-Purpose Hypertext Abstract Machine*, Proc. of ACM Hypertext'87*, Chapel Hill, NC, November, pp. 21-32, 1987.

[46] M. Bieber, H. Oinas-Kukkonen, V. Balasubramanian, Hypertext Functionality, *ACM Computing Surveys Symposium on Hypertext and Hypermedia*, 2000.

[47] H. Oinas-Kukkonen, Flexible CASE and Hypertext, *ACM Computing Surveys Symposium on Hypertext and Hypermedia*, 2000.

[48] K. Østerbye, Literate Smalltalk Programming Using Hypertext, *IEEE Transactions on Software Engineering*, 21(2):138-145, 1995.

[49] A. Dattolo, V. Loia, Active Distributed framework for adaptive hypermedia, *International Journal of Human-Computer Studies*, 26:605-626, 1997.

[50] M. Amsellem, ChyPro: A Hypermedia Programming Environment for Smalltalk-80, *Proc. of ECOOP'95*, Aarhus, Denmark, August, 1995.

[51] M. L. Creech, D. F. Freeze, M. L. Griss, Using Hypertext in Selecting Reusable Software Components, *Proc. of ACM Hypertext'91*, San Antonio, TX, December, pp. 25-38, 1991.

[52] I. Goldstein, D. Bobrow, A Layered Approach to Software Design, in D. Barstow, H. Shrobe, E. Sandewell (eds.), *Interactive Programming Environments*, McGraw Hill, pp. 387-413, 1984.

[53] D. Hicks, *A Version Control Architecture for Advanced Hypermedia Environments*, Dissertation. Department of Computer Science, Texas A&M University, College Station TX, 1993.

[54] A. Dattolo, A. Gisolfi, Analytical Version Control Management in a Hypertext System, *Proceedings of the 3th International Conference on Information and Knowledge Management*, November 29 – December 1 1994, Caithersburg, MD, USA, pp. 132-139, 1994.

[55] D. Durand, A. Haake, D. Hicks, F. Vitali. (eds.), *Proceedings of the ECHT '94 Workshop on Versioning in Hypertext Systems*, Technical Report of the Computer Science Department, Boston University, 95-01, http://www.cs.bu.edu/techreports/95-001/Home.html and Arbeitspapiere der GMD 894, GMD-IPSI, Darmstadt, Germany, 1994.

[56] D. Hicks, A. Haake, D. Durand, F. Vitali (eds.), *Proceedings of the ECSCW '95 Workshop on the Role of Version Control in CSCW Applications*, Technical Report of the Computer Science Department, Boston University, 96-06, 1995, http://www.cs.bu.edu/techreports/96-009-ecscw95-proceedings/Book/proceedings_txt.html

[57] J. E. Whitehead, A Proposal for Versioning Support for the Chimera System, *ECHT '94 Workshop on Versioning in Hypertext Systems*, Edinburgh, UK, 1994.

[58] A. Dattolo, V. Loia, Collaborative Version Control in an Agent-based Hypertext Environment, in *Information Systems*, 21(2):127-145, 1996.

[59] A. Dattolo, V. Loia, Distributed information and control in a concurrent hypermedia-oriented architecture, *International Journal of Software Engineering and Knowledge Engineering*, 10(3):345-369, 2000.

[60] XML – Extensible Markup Language, http://www.w3.org/XML/

[61] F. Vitali, D. Durand, Using Versioning to Provide Collaboration on the WWW, *The World Wide Web Journal* 1(1), O'Reilly, pp. 37-50, 1995.

[62] D. Raggett, A. Le Hors, I. Jacobs, *HTML 4.0 Specification*, W3C Recommendation, 1998, http://www.w3.org/TR/PR-html40/

[63] J. Whitehead, M. Wiggins, WEBDAV: IETF Standard for Collaborative Authoring on the Web, *IEEE Internet Computing* September/October 1998, pp. 34-40, 1998.

[64] T. Ellison, WEBDAV Versioning scenario, *INTERNET-DRAFT draft-ietf-deltav-scenarios-00.1* February 10, 2000, http://www.webdav.org/deltav/scenarios/scenarios-00-1.htm

[65] DASL (DAV Searching & Locating Working Group). http://www.webdav.org/dasl/

[66] J. Whitehead, The Future of Distributed Software Development on the Internet. From CVS to WebDAV to Delta-V,1999, http://www.webtechniques.com/archives/1999/10/whitehead/.

[67] D. L. Hicks, J.J Leggett, P. J. N⌉rnberg, J. L. Schnase, A Hypermedia Version Control Framework, *ACM Transactions on Information Systems*, 16(2):127-160, 1998.

[68] C. Reichenberger, Delta Storage for Arbitrary Non-Text Files, *Proc. of the 3rd International Workshop on Software Configuration Management*, Trondheim, Norway, June 12-14, 1991.

[69] D. B. Leblang, R. P. Jr. Chase, H. Spilke, Increasing Productivity with a Parallel Configuration Manager, *Proc. of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, January 27-29, 1988, Teubner Verlag, January 1988.

[70] D. Lubkin, Heterogeneous Configuration Management with DSEE, *Proc. of the 3rd International Workshop on Software Configuration Management, Trondheim*, Norway, June 12-14, 1991.

[71] D. B. Leblang, R. P. Jr. Chase, Computer-Aided Software Engineering in a Distributed Workstation Environment, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, ACM SIGPLAN Notices, 19(5), May 1984.

[72] J. Estublier, S. Ghoul, S. Krakowiak, Preliminary Experience with a Configuration Control System for Modular Programs, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* May 1984, ACM SIGPLAN Notices, 19(5), May 1984.

[73] J. Estublier, S. Dami, M. Amiour, High Level Process Modelling for SCM Systems, *Proc. of the 7th International Workshop on Software Configuration Management*, Boston, Massachusetts, May 18-19, 1997*, Springer Lecture Notes in Computer Science*, No. 1235.

[74] U. Asklund, B. Magnusson, A. Persson: Experiences: Distributed Development and Software Configuration Management, *Proc. of the 8th International Symposium on System Configuration Management*, Brussels, Belgium, July 20-21, 1998, *Springer Lecture Notes in Computer Science*, No. 1439.

[75] L. Wingerd, C. Seiwald, High-Level Best Practices in Software Configuration Management, *Proc. of the 8th International Symposium on System Configuration Management*, Brussels, Belgium, July 20-21, 1998*, Springer Lecture Notes in Computer Science*, No. 1439.

[76] L. Bendix, Continuous Improvement of the Configuration Management Process, *Proc. of the conference Views on Software Development in the New Millennium*, Reykjavik, Iceland, August 31 - September 1, 2000.

[77] J. Estublier, J.-M. Favre, P. Morat, Towards SCM/PDM Integration?, *Proc. of the 8th International Symposium on System Configuration Management*, Brussels, Belgium, July 20-21, 1998, *Springer Lecture Notes in Computer Science*, No. 1439.

[78] J.-O. Larsen, H. M. Roald, Introducing ClearCase as a Process Improvement Experiment, *Proc. of the 8th International Symposium on System Configuration Management*, Brussels, Belgium, July 20-21, 1998, *Springer Lecture Notes in Computer Science*, No. 1439.