A Software Configuration Management Model for Supporting Component-Based Software Development

Hong Mei, Lu Zhang, Fuqing Yang
Department of Computer Sci. & Tech., Peking University, Beijing 100871, PRC
<{meih, zhanglu, yang}@cs.pku.edu.cn>

Abstract

Software configuration management (SCM) is viewed as an important key technology in software development and is being adopted in software industry more and more widely. And component-based software development (CBSD) is an emerging paradigm in software development. However, the traditional SCM method can not support CBSD effectively and efficiently. In this paper, we analyze the objects that need to be managed in CBSD and present a component-based SCM model to improve CBSD with effective SCM support. In this model, components, as the integral logical constituents in a system, are managed as the basic configuration items in SCM, and relationships between/among components are defined and maintained, in order to support version control of large objects and logical modeling of system architecture. Also some key issues in the model are discussed and an SCM system that supports this model is introduced.

Keywords: software configuration management, software reuse, software component, version control

1. Introduction

Software configuration management (SCM) is an important field in software engineering research. It is a disciplined approach to manage the evolution of software development, maintenance practices and the software products [1]. SCM manages the artifacts in the developing process, controls the changes in the software and its components, and assists the software development.

For a long time, SCM has been the focus of software engineering research and a great amount of research has been carried out on SCM. As a result, a lot of research fruits have been achieved, and SCM becomes more and more mature and has been successfully put into practice. In fact, there exist a lot of SCM products already in use. The typical representatives are ClearCase [2], TRUEChange [3], and PVCS [4].

With the research on and application of SCM, more and more researchers realize that the most effective and reliable technology available for meeting software development demands is SCM [1]. More and more software development organizations have noticed the importance of SCM, and viewed SCM as their infrastructure for software development. As a result, the competition among SCM suppliers becomes more fierce and the SCM revenue is estimated to reach one billion dollars by the end of this year [1].

In the previous research, eight areas of functionality of SCM systems were found: version control, configuration support, team support, change control, build support, process control, status reporting, and audit control [1] [5] [6]. These functional areas mainly cover the management issues of software development. To provide these eight areas of functionality, different SCM systems use different models, such as the checkout/checkin model, the composition model, the long transaction model, and the change set model [7]. In recent years, the focus of SCM research is on software process support in SCM systems [8] [9], distributed configuration management systems [10] [11], unified version models [12] [13], and integration of SCM and PDM [14] [15].

Component-based software development (CBSD) is an emerging paradigm in software development, which may change the way we develop software greatly [16]. The aim of CBSD is to develop new software by widely reusing pre-fabricated software components. To develop software in this way, our tasks are divided into two categories. The first one is to develop reusable components from scratch and/or modify existing components. The second one is to compose components to form a system, a sub-system, or a bigger component that can be called a composite component. Up to now, the research is mainly focused on the technical issues in CBSD, such as software architecture [17] and distributed object [18][19][20]. However, the management issues in CBSD have been neglected.

Like the traditional way to develop software, CBSD also needs the support of SCM. In the traditional SCM, primitive configuration items are files. The version control of files is the basis of SCM, and relationships and configurations are set up on files. However, CBSD brings new challenges to SCM and the traditional SCM methods and systems can not meet these challenges satisfactorily. Some identified issues are as follows:

1) In CBSD, usually an application is implemented into many files (for example, more than five hundred). It is very difficult for developers to manage so many files and their relationships.

2) A file is not a logical constituent in an application. Usually, a logical constituent is implemented into a set of files. It is difficult to manage a logical constituent, such as a component in CBSD, as a whole.

3) Software architecture has been viewed as an important milestone in the lifecycle of software. Software architecture can be defined by components, connectors, and their configuration, among which components and connectors are the logical constituents in an application and configuration describes the logical relationships between components and connectors. It is unnatural, and even difficult, to maintain the logical architecture of an application when it is still developing with the traditional SCM techniques.

We can, of course, use the traditional SCM to manage logical constituents and maintain the logical relationships between/among constituents. But it is unnatural and inefficient, because the developers must bear in their mind the logical constituents and the logical relationships in order to keep and trace the links inside the constituents and between the constituents.

In our opinion, using files as the primitive items and asking developers to operate on the files directly are not an efficient way to support CBSD with SCM. We argue that a better way is to treat the basic logical constituents in a system as the primitive items, which are to be managed by SCM. To support CBSD, SCM should solve the following issues:

1) Viewing each component as an entity and operating on components. The primitive configuration items should be components, which may be implemented into a set of files. And instead of managing every file in a component individually, we should treat the component as a whole. The evolutionary history of each component should be recorded by SCM. With the support of SCM, any version of any component can be easily retrieved, and the consistency and completeness of each component could be maintained easily. SCM should also provide additional information for the development of the components.

2) Controlling the concurrent modifications to each component. A component could be modified simultaneously. SCM should control this kind of concurrent development effectively and efficiently, thus avoiding or solving the possible conflicts in the development.

3) Managing component composition and relationships between/among components. Small components can be composed to form bigger components. SCM should control and manage the process of component composition and its products. SCM should also manage various relationships between/among components. In this way, the developers can maintain the logical architecture of an application effectively and efficiently.

In this paper, we analyze the objects that need to be managed in CBSD. Based on the analysis, a component-based SCM model is presented. We also discuss some key issues in the model, which must be considered while putting this model into practice. At last, a SCM system that supports this model is introduced.

2. The Component-based SCM Model

2.1 Basic concepts

The goal of this model is to support CBSD more efficiently, so some concepts in CBSD are still used in this model, and there are a few changes in some concepts concerning the traditional SCM. Here some important concepts are explained.

File: In this model, files are the basic physical storage units. A file does not represent any logical constituent in a system. Usually, a group of files represent a logical constituent.

Component: The concept of components is very central in CBSD. A component is an integral logical constituent in a system, which is relatively independent on its surroundings concerning the functions and behaviors. In CBSD, there are two categories of components to be managed. The first are primitive components and the second are composite components. In this model, primitive components, which are usually implemented into a set of files, are the primitive configuration items, and composite components are represented by configurations.

Configuration and configuration item: In this model, configurations are used to represent composite components. A configuration is composed of a set of configuration items. Each configuration item is a primitive component or an existing configuration. A configuration as a configuration item in another configuration is called a sub-configuration.

Baseline: In this model, baselines stand for the boundaries of composite components. A new baseline of a configuration can be created by selecting a version or a baseline of each configuration item in the configuration.

Relationship: There are relationships among components. For example, if one component depends on another component, there is a relationship between them. In this model, relationships are recorded and used to manage components.

2.2 The objects that need to be managed

Primitive component

Primitive components are the basic logical units in CBSD. A primitive component can be quite complex. It is usually implemented as a set of files. However, the files in a primitive component usually evolve as a whole. So, only when the evolution of individual files are transparent to developers can the evolution of the whole primitive component be seen. A primitive component can have several evolutionary directions, and all the evolutionary directions should be under the control of SCM. Usually, a primitive component is under the development or modification of a team. It is also important to manage and control the parallel development of a primitive component.

The operations for management of primitive components are:

1) Checkout and checkin. The checkout-edit-checkin style is used to manage each evolutionary direction of a primitive component. To make a change in a primitive component, it should be checked out first. After the change, it should be checked in and thus becomes a new version.

2) Branching and merging. Primitive components can have several evolutionary directions. By using the branching operation, a new evolutionary direction can be created from any of the existing versions of a primitive component. An evolutionary direction is also called a branch. Different branches can merge together during the evolution of the primitive component.

3) Concurrency. A primitive component is usually under the development of a team. The concurrency operation is used to coordinate the different modifications from different developers in order to form a primitive component.

Composite component

Composite components are constructed by the composition of some other primitive and composite components in CBSD. With the changes in requirements, composite components also evolve after the composition. SCM should manage both the construction and evolution of composite components. A composite component can also have several evolutionary directions, and SCM should manage all the evolutionary directions.

The operations used to manage composite components can be represented by the operations on configurations. They are:

1) Configuration creation. A new configuration is created by selecting some primitive components and some existing configurations.

2) Baseline creation. A composite component will evolve while the components which form the composite component evolve. A new baseline for the configuration is created by selecting a version for each primitive component and a baseline for each sub-configuration in the configuration. This operation is called baseline creation. A baseline stands for a milestone version of a composite component.

3) Configuration modification. What a composite component is composed of can also be changed. Modification to a configuration represents this kind of change.

4) Configuration branching. A composite component can also have several evolutionary directions. The branching operation of a configuration can be used to manage all the evolutionary directions of the corresponding composite component.

Relationship between/among components

The relationships between/among components can also be treated as a kind of objects that need to be managed. The operations to manage relationships between components are creation and tracing.

1) Creation. Relationships can be created in SCM to represent the corresponding relationships between/among components.

2) Tracing. By tracing these relationships, some useful information for the management of the components can be achieved. For example, the potential impact of a change can be determined by analyzing the dependencies associated with the components that intend to change.

2.3 The model

Based on the above analysis, we can present the component-based SCM model. The model is shown as Figure-1.

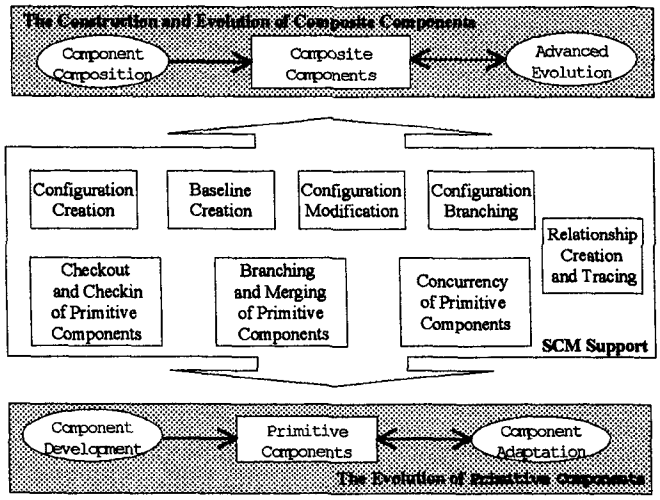

Figure-1 Component-Based SCM Model

At the bottom of the figure is the evolution of primitive components. On top of the figure is the construction and evolution of composite components. Both of them should be under the control of SCM. The relationships are objects under management too.

The formal definitions of primitive components, configurations, and relationships in this model are as follows:

Primitive Component

Definition 1 (Versioned File): F is a versioned file, if and only if, F is a directed tree F=(V,E); where V={v | v is a version of F}, and E={(v_1,v_2)|$v_1,v_2 \in V$, v_2 is the next version of v_1 or v_2 is branched from v_1}.

From the above definition, v is a version of F=(V,E), if and only if $v \in V$.

F_1 and F_2 are independent, if and only if, for any v being a version of F_1, v is not a version of F_2; and for any u being a version of F_2, u is not a version of F_1.

A file set FS is valid, if and only if, for any F_1, $F_2 \in FS$, F_1 and F_2 are independent.

Thus, F' is evolved from F (noted as $F \rightarrow F'$), if and only if, when F=(V,E) and F'=(V',E'), there exist u and v, where u is a version of F, and v is not a version of F, and $V'=V \cup \{v\}$, and $E'= E \cup \{(u, v)\}$.

For two file sets FS and FS', FS' is evolved from FS (noted as $FS \rightarrow FS'$), if and only if, FS and FS' are both valid, and $FS \neq FS'$, and for any $F_1 \in FS$, there exists $F'_1 \in FS'$, $F_1 \rightarrow F'_1$ or $F_1=F'_1$, and for any $F'_2 \in FS'$, there exists $F_2 \in FS$, $F_2 \rightarrow F'_2$ or $F_2=F'_2$.

Definition 2 (Version of a Primitive Component): cv is a version of a primitive component, if and only if, cv is a set of file versions; and there exists a set of file FS, for any $v \in cv$, there exists $F \in FS$, and v is a version of F; and for any $F \in FS$, there exists $v \in cv$, and v is a version of F.

A primitive component version cv is on a valid file set FS, if and only if, for any $v \in cv$, there exists $F \in FS$, and v is a version of F; and for any $F \in FS$, there exists $v \in cv$, and v is a version of F.

It is easy to prove, that, for two valid file sets FS and FS', and cv as a primitive component version on FS, if $FS \rightarrow FS'$, cv is also on FS'.

For two primitive component versions cv_1 and cv_2, cv_2 is the next version of cv_1 or cv_2 is branched from cv_1, if and only if, there exist four valid file sets: FS_1, FS_2, FS_3 and FS_4, which satisfy,

1) $cv_1 \neq cv_2$;
2) $FS_1 \cup FS_2 \cup FS_4$ and $FS_1 \cup FS_3 \cup FS_4$ are valid;
3) $FS_2 \rightarrow FS_3$ or $FS_2 = FS_3$;
4) cv_1 is on $FS_1 \cup FS_2$, and is also on $FS_3 \cup FS_4$;
5) for any $u \in cv_1$, if there exists $F \in FS_2$, and u is a version of F, there exists $v \in cv_2$, and $F' \in FS_3$, and v is a version of F', and if $u \neq v$, v is not a version of F, and v is the next version of u or v is branched from u.

Definition 3 (Versioned Primitive Component): Com is a versioned primitive component, if and only if, Com is a directed tree Com=(CV,CE); where CV={cv | cv is a version of Com}, and CE={(cv_1,cv_2)|$cv_1,cv_2 \in CV$, cv_2 is the next version of cv_1 or cv_2 is branched from cv_1}.

For a primitive component version cv and a primitive component Com, cv is a version of Com=(CV,CE), if and only if $cv \in CV$.

Thus, Com' is evolved from Com (noted as Com → Com'), if and only if, when Com=(CV, CE) and Com'=(CV',CE'), there exist cu and cv, cu is a version of Com, cv is not a version of Com, and $CV'=CV \cup \{cv\}$, and $CE'= CE \cup \{(cu,cv)\}$.

Two primitive component Com_1 and Com_2 are independent, if and only if, for any cv_1 being a version of Com_1, and any cv_2 being a version of Com_2, and two valid file sets FS_1 and FS_2, if cv_1 is on FS_1, and cv_2 is on FS_2, $FS_1 \cup FS_2$ is valid.

A primitive component set CS is valid, if and only if, for any $C_1, C_2 \in CS$, C_1 and C_2 are independent.

Configuration

Definition 4 (Configuration): Conf is a configuration on a valid primitive component set CS, if and only if, $Conf=CIS_{CS} \cup CIS_{Conf}$; where CIS_{CS} is a set of primitive components, and for any $Com \in CIS_{CS}$, $Com \in CS$, and CIS_{Conf} is a set of configurations on CS.

For a configuration Conf on a valid primitive component set CS, any $Item \in Conf$ is called a configuration item.

Definition 5 (Baseline): For a configuration $Conf=CIS_{CS} \cup CIS_{Conf}$ on a valid primitive component set CS, the set $Baseline=Baseline_{CS} \cup Baseline_{Conf}$ is called a baseline of Conf, if and only if, for any $b \in Baseline_{CS}$, there exists $Com \in CIS_{CS}$, and b is a version of Com; and for any $b_1, b_2 \in Baseline_{CS}$, $Com \in CIS_{CS}$, if $b_1 \neq b_2$, either b_1 or b_2 is not a version of Com; and for any $b \in Baseline_{Conf}$, there exists $SubConf \in CIS_{Conf}$, and b is a baseline of SubConf, and for any $b_1, b_2 \in Baseline_{Conf}$, $SubConf \in CIS_{Conf}$, if $b_1 \neq b_2$, either b_1 or b_2 is not a baseline of SubConf.

Relationship

Definition 6 (Relationship): Rel is a relationship on a primitive component set CS, if and only if, $Rel=\{c | c \in CS$ or c is a configuration on CS$\}$.

3. Some Key Issues in the Model

3.1 Consistency maintenance

In the traditional SCM models, the objects to be managed are not the logical constituents in a system. SCM need not maintain the consistency of the objects. In our model, the objects are mainly components. It is necessary to maintain the consistency of the components.

Consistency of primitive component

For a primitive component, the consistency means that any version of the component should logically be a component. Checking the consistency of a primitive component is specific to the component model applied in CBSD. This kind of consistency checking should be performed whenever a new version of a primitive component is checked in.

Consistency of configuration

For a configuration, the consistency has two meanings. The first is that, in any baseline of the configuration, the components should be logically composed. The second is that logically any baseline of the configuration should be a component. Checking the consistency taking the first meaning can be performed by analyzing the corresponding component descriptions. Checking the consistency taking the second meaning is specific to the component model applied in CBSD. Both consistency checking should be performed whenever a new baseline is created.

Consistency between two neighboring versions/baselines

For a component is a logical constituent in a system, any neighboring versions of a primitive component and any neighboring baselines of a configuration should conform to the versions of the same requirement and design specifications for the corresponding components. It is also important to check the consistency between two neighboring versions/baselines whenever a new version/baseline is created.

3.2 Concurrency control strategy of primitive component development

It is very important to manage the concurrent modifications to one primitive component. A primitive component is composed of a set of files. It is apparently necessary to allow different developers to modify different files in a primitive component concurrently.

In SCM, concurrency control is realized by the operating modes of files. There are three modes: read-only, exclusive-write, and share-write. Only when a developer has the file in the mode of exclusive-write or share-write, can he make changes to the file. Using share-write mode allows more room for concurrency, but it relies on the automatic merging of two file versions, and the validity of this kind of merging can not be guaranteed.

Here, we propose a concurrency control strategy for primitive component development without using the share-write mode. A primitive component can be checked out by several developers. When a primitive component is checked out, the operating modes of all the files in the component are set to the read-only mode. When a developer wants to make a change to a file in the checked out primitive component, he changes the operating mode of the file to exclusive-write and makes the actual change. In this way, different developers can make changes to different files without affecting each other. When one developer wants to make a change to a file that another developer is currently making a change to, he will not be allowed to change the operating mode of this file to exclusive-write and thus can not make the change. So, the checkout-edit-checkin style of this model is a little bit more complex than the traditional one. It is shown in Figure-2.

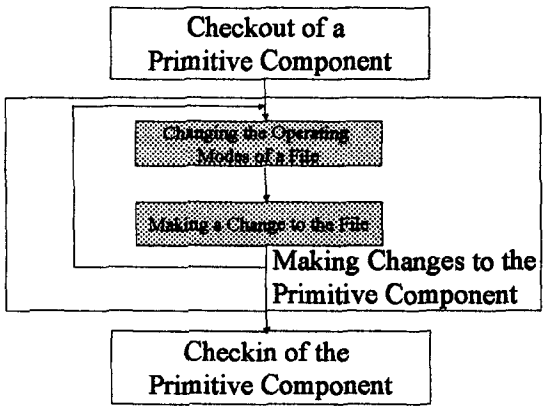

Figure-2 New Checkout-Edit-Checkin Style

3.3 Horizontal and vertical merging

For a primitive component is composed of files, merging of different versions of a primitive component is different from merging of different

versions of a file. This kind of merging includes two aspects. The first aspect is the merging of structure. The merged primitive component version should be composed of every file that is in either of the original version. The second aspect is the merging of files. If a file is in both of the primitive component versions but since different primitive component versions have different file versions, the file version in the merged primitive component version should be the merging of the two original file versions. This kind of merging takes place between two different branches. So, we call it horizontal merging in this paper. An example of horizontal merging is shown in Figure-3 (left).

For any change to any of the files in a primitive component is a change to the whole primitive component, the version tree of the primitive component will grow very fast. It is necessary to merge several neighboring versions on a branch into a single version. We call it vertical merging in this paper. This kind of merging is used to manage and control the growth of the version tree of a primitive component. An example of vertical merging is shown in Figure-3 (right).

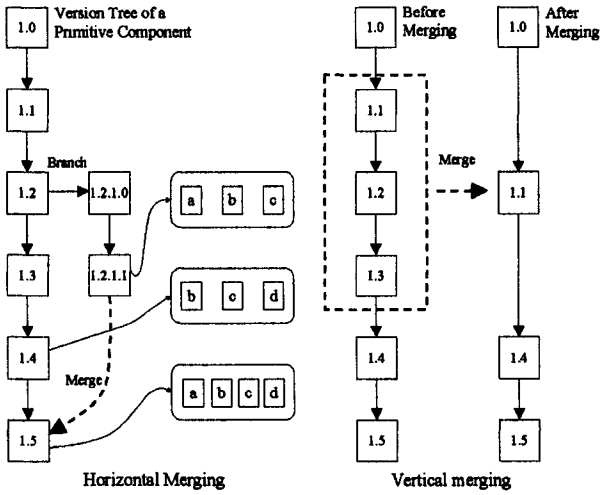

Figure-3 Horizontal and Vertical Merging

4. JBCM - an SCM System Conforming to the Model

We have developed a SCM system, called JBCM, to support the proposed SCM approach. The architecture of JBCM is shown in Figure-4. The system is divided into three parts. The first part is the mechanism of version control and concurrency control of primitive components. This part is the basis of the whole system, and it is transparent to the users. Based on this part is the part of the basic management functions, which support the basic management of primitive components, composite components, and relationships between them. Actually the issues discussed in the second section of this paper only cover the basic functional areas in SCM. In fact, there are several other functional areas in SCM, which are called high-level SCM functional areas. These functional areas include change control, build support, process control, status reporting, and audit control. The last part of JBCM is the high-level management that covers the high-level functional areas.

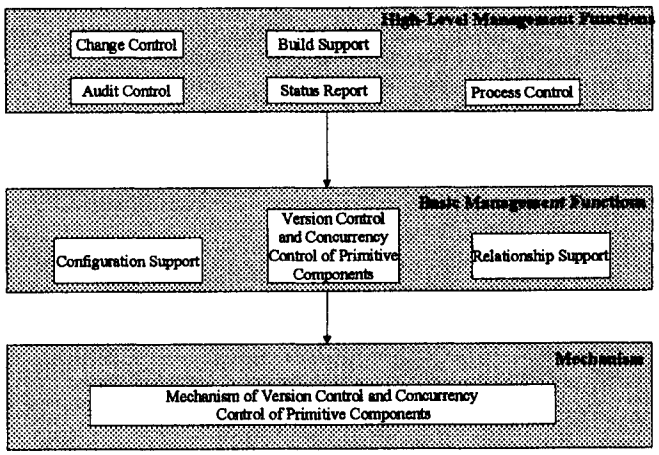

Figure-4 JBCM System

5. Related Work

Version control of large objects

In CVS [21], module is the main unit of version control operations. A module is composed of a set of files. Users are mainly dealing with modules rather than files, but the history of a module can not be recorded automatically. In COOP/Orm [22], the version control unit is a document. A document is composed of files and directories. COOP/Orm maintains a version tree of each document rather than several version trees of files in the document. In PRCS [23], the version control unit is a project. A project in PRCS is similar to a document in COOP/Orm.

The research mentioned above mainly addresses the technology of version control for large objects. This technology is helpful to manage large-scale software development. Primitive components are also large objects, and the technology, to some extent, can be used for the version control of primitive components. However, the technology can not be used to manage both the history and concurrent development of a primitive component efficiently, and the version control of primitive components is only a SCM issue in CBSD. In this paper, our approach includes a better solution to the version control of primitive components.

System modeling in SCM at the architecture level

In [24], it is argued that system modeling should be at the architecture level, and the basic elements in system modeling should be logical items in the development process. The logical items are under version control. In [25], it is suggested that architecture description language (ADL) should be introduced into SCM to enhance the system modeling ability of SCM.

The research mentioned above is helpful to manage the architecture of software. To some extent, it can be used to manage the component composition in CBSD. However, it is far from solving all the SCM issues in CBSD. In this paper, our approach covers almost all the SCM issues in CBSD.

5. Conclusion

In this paper, we focus on the SCM issues in CBSD, and present a component-based SCM model to support CBSD effectively and efficiently. In our model, primitive components are the primitive configuration items, composite components are represented by configurations, and relationships are defined between/among primitive components and configurations. The evolutionary history of each primitive component is managed, the concurrent

modifications to one primitive component are controlled, and configurations and baselines are used to manage the construction and the evolution of composite components. Some key issues such as consistency maintenance, concurrency control strategy, horizontal and vertical merging are considered too. Based on this model, a SCM system JBCM is introduced which is an automated tool to perform SCM in CBSD according to the model. Our approach supports the version control of large objects and the logical modeling of system architecture, also supports the high-level SCM functional areas.

The future tasks of research include supporting component merging at the syntax level, improving the ability of system modeling at the architecture level, and supporting logical versions of components.

Acknowledgements

This effort gets support from the key project in the State 9th Five-year Plan and State 863 High-Tech Program.

References

[1] C. Burrows, G. George, and S. Dart, Configuration Management, Ovum Ltd., 1996.

[2] Rational Corporation, ClearCase Concepts Manual. Document Number 800-010043-000, January 1998.

[3] McCabe & Associates Corporation, McCabe TRUEChange, Available WWW <URL: http://www.mccabe.com/products/truechange.htm> (2000)

[4] Merant Corporation, PVCS Product, Available WWW <URL: http://www.merant.com/products/pvcs/> (2000)

[5] S. Dart. Spectrum of Functionality in Configuration Management Systems, Technical Report CMU/SEI-90-TR-11, Software Engineering Institute, Pittsburgh, Pennsylvania, December 1990.

[6] S. Dart, Concepts in Configuration Management Systems, In Proceedings of the Third International Workshop on Software Configuration Management, pages 1-18. ACM SIGSOFT, 1991.

[7] P. Feiler, Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Pittsburgh, Pennsylvania, March 1991.

[8] J. Estublier, S. Dami, and M. Amiour. High Level Process Modeling for SCM Systems, In Proceedings of the Seventh International Workshop on Software Configuration Management, New York, New York, 1997.

[9] D.B. Leblang, Managing the Software Development Process with ClearGuide, In Proceedings of the Seventh International Workshop on Software Configuration Management, New York, New York, 1997.

[10] J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy, Distributed Configuration Management via Java and the World Wide Web, In Proceedings of the Seventh International Workshop on Software Configuration Management, New York, New York, 1997.

[11] B. Milewski, Distributed Source Control System, In Proceedings of the Seventh International Workshop on Software Configuration Management, New York, New York, 1997.

[12] R. Conradi and B. Westfechtel, Towards a Uniform Version Model for Software Configuration Management, In Proceedings of the Seventh International Workshop on Software Configuration Management, New York, New York, 1997.

[13] A. Zeller and G. Snelting, Unified Versioning through Feature Logic, ACM Transactions on Software Engineering and Methodology, 6(4):398--441, October 1997.

[14] J. Estublier, J. Favre, and P. Morat, Toward SCM/PDM Integration? In Proceedings of the Eighth International Symposium on System Configuration Management, Brussels, Belgium, 1998.

[15] B. Westfechtel, and R. Conradi, Software Configuration Management and Engineering Data Management: Differences and Similarities. In Proceedings of the Eighth International Symposium on System Configuration Management, Brussels, Belgium, 1998.

[16] M. Aoyama, Component-Based Software Engineering: Can it Change the Way of Software Development? In Proceedings Volume II of the 1998 International Conference on Software Engineering. April 1998.

[17] M. Shaw, and D. Garlan, Software Architecture, Prentice Hall, 1996.

[18] Microsoft Corporation, COM: Delivering on the Promises of Component Technology, Available WWW <URL: http://www.microsoft.com/com/> (2000)

[19] Object Management Group, Common Object Request Broker Architecture (CORBA) Version 2.3, Available WWW <URL: http://www.omg.org> (1999)

[20] A. Thomas, Enterprise JavaBeans: Server Component Model for Java Platform, White Paper, Available WWW <URL http://www.javasoft.com/products/ejb/>(2000)

[21] B. Berliner, CVS II: Parallelizing software development, In Proceedings of the Winter 1990 USENIX Conference, January 22-26, 1990, Washington, DC, USA (Berkeley, CA, USA, Jan. 1990), USENIX Association, Ed., USENIX, pp. 341-352.

[22] B. Magnusson, and U. Asklund, Fine Grained Version Control of Configurations in COOP/Orm, In Proceedings of the Sixth International Workshop on Software Configuration Management, Berlin, Germany, March, 1996.

[23] J. MacDonald, P.N. Hilfinger, and L. Semenzato, PRCS: The Project Revision Control System, In Proceedings of the Eighth International Symposium on System Configuration Management, Brussels, Belgium, July 1998.

[24] H.B. Christensen, Experiences with Architectural Software Configuration Management in Ragnarok, In Proceedings of the Eighth International Symposium on System Configuration Management, Brussels, Belgium, July 1998.

[25] A. van der Hoek, D.Heimbigner, and A.L.Wolf, System Modeling Resurrected, In Proceedings of the Eighth International Symposium on System Configuration Management, Brussels, Belgium, July 1998.