A Configuration Management System Supporting Component-Based Software Development

Lu Zhang[1], Hong Mei[2], Hong Zhu[3]
[1]Department of Computer Science, The University of Liverpool, Liverpool L69 3BX, UK
Email: lzhang@csc.liv.ac.uk
[2]Department of Computer Science and Technology, Peking University, Beijing 1000871, P. R. China
Email: meih@cs.pku.edu.cn
[3]School of Computing and Mathematical Sciences,Oxford Brookes University, Oxford, OX3 0BP, UK
Email: hzhu@brookes.ac.uk

Abstract

Component-based software development has been viewed as an emerging paradigm of software development. This paper analyzes the requirements of configuration management in component-based development process. Based on the analysis, a prototype configuration management system is proposed to meet the requirements. An example of using the system is also given.

Keywords: component-based software development, configuration management, software component, software architecture.

1. Introduction

Configuration management (CM) is a disciplined approach to managing the evolution process of software development and maintenance [5]. It manages the artifacts produced in the development process, controls the changes to the software and its components, keeps tracks of evolution, and thus assists software development. For a long time, CM has been a focus of software engineering research. In software development practices, there has been a growing realization that CM is one of the most effective and reliable technologies available for meeting software development demands [5]. More and more organizations in software industry have recognized the importance of CM, and viewed CM systems as a part of their infrastructure of software development.

Component-based software development (CBSD) is an emerging paradigm of software development [1]. CBSD originates from the research on software reuse and software component technology (see e.g. [10] and [17]). Up to now, there has been intensive research reported in the literature surrounding this new paradigm of software development, such as component models, component description languages, commercial-off-the-shelf (COTS) components, distributed objects, software architecture, software frameworks, and design patterns.

However, research on CM support for CBSD seems to be ignored, and therefore there are very few reports that are related to both CM and CBSD in the literature. In [6] and [16], it is suggested that architecture description languages (ADL) should be incorporated into CM systems to enhance their ability of system modeling.

In this paper, we analyze the requirements of CM for CBSD, and present a prototype CM system to meet the requirements. Both the concept model of the system and the design and implement issues of the system are presented in details. An example is also given to shown how to use the system.

2. New Requirements of CM for CBSD

2.1. An Overview of CBSD

As described in [4], the aim of CBSD is to develop new software systems by widely reusing pre-fabricated software components. Therefore, the development tasks can be divided into two categories. The first is to develop reusable components from scratch and/or by modifying existing components. Components of this kind are usually called *primitive components*. The second is to compose the existing components to form the composed system and/or bigger components that can be used to form the composed system eventually. A component composed of other components is usually called a *composite component*.

However, the research on software architecture has shown, that when composing a new system, it may reduce the reusability of components to connect components directly with each other. Therefore, connectors are needed to serve as glue between components, and the architecture description of how the connectors and the components are connected is also needed. Connectors can also be either primitive or composite. In this paper, as we think components and connectors can be viewed as the same from the perspective of CM, we use the term ‘constituent’ to represent a component or a connector.

Generally speaking, the main differences between CBSD and the conventional way of software development are as follows:

1) Different ways of system construction. In the conventional way of software development, the system is usually implemented by a group of source code files, which can be compiled and linked together; while in CBSD, system construction is a recursive process, in which, primitive constituents are used to construct composite constituents, both primitive and composite constituents are used to construct larger composite constituents, and the system is constructed from the primitive constituents, and the previously constructed composite constituents.

2) Different languages for system construction. In the conventional way of software development, programming languages are used to implement the system. In CBSD, programming languages are used to implement the primitive constituents, and component description languages and architecture description languages to construct the composite components and the system. In a programming language, procedure calls are used to connect different parts, but description languages directly describe the inter-connection between components. As CM is concerned primarily with maintaining the integrity of software systems and the interoperability of components as they undergo change [12], these differences may incur some new requirements of CM for CBSD.

2.2. Version Management of Constituents

As the development of primitive constituents and the development of composite constituents are quite different, the ways of management of versions of these two types of constituents should also be different. A primitive constituent, which is usually implement by a group of files, is the basic unit of a system. It should be managed as a whole and every version should be recorded while it undergoes change. In contrast, a composite constituent is the composition of other constituents. Its version management should be based on the version management of its own constituents. In the past a few years, some techniques have been developed separately to support relationships between primitive objects (such as [13]) and version controls of composite objects (such as [9]). However, there is lack of a model and technique that can support the version management of both types of constituents in CBSD.

2.3. Integrity Maintenance

In CBSD, the maintenance of integrity that CM should consider has two aspects: checking the consistency and analyzing the impact of change.

- **Checking consistency**

The consistency of a primitive constituent is the consistency between different parts of the primitive constituent, such as the consistency between the head and the body, the consistency between the specification and the implementation. This may be achieved via analyzing the source code and description of the constituent.

The consistency of a composite constituent is also the consistency between different parts of the constituent. However, the parts themselves are the constituents. Actually, there are two kinds of consistency of a composite constituent. The first is syntactic consistency, which may be checked through parsing the descriptions of its constituents. The second is semantic consistency, which reflects whether its constituents can match each other to provide the correct functions. We think semantic consistency of a composite constituent still needs further investigation.

- **Change affection analysis**

Generally speaking, existing research results on change affection analysis fall into two classes, the formal and informal approaches. The formal approach, such as [14] and [11], calculates the dependence between configuration items from the information contained in the item or its description, such as from the source code. The informal approach, such as [2] and [8], uses predicted dependence relationships for change affection analysis. These techniques are still applicable to the primitive constituents. However, new affection analysis method should be developed to provide more effective support to the analysis of composite constituents. In fact, for a composite constituent, affection analysis may be analyzed according to the architectural description because the change of an element of a composite constituent will affect the elements directly connected to the changed element. Such effect can be localized to the directly linked elements because proper revision of the affected element should be able to maintain the interfaces with other elements.

3. Concept Model

To meet the requirements, we designed and implemented a prototype CM system named JBCM (the abbreviation for Jade Bird Configuration Management). The underlying idea of our JBCM system is rather simple. First, we view each primitive constituent as the basic unit, and record version history of it rather than each file in it; second, we use recursive defined configurations to represent composite constituents and therefore can maintain versions of composite constituents; third, we use dependence relationships to assistant change affection analysis; and finally, we exploit a new strategy to manage the concurrent modifications to files in one primitive constituent. In this section, we present the concept model based our idea.

3.1. Overview

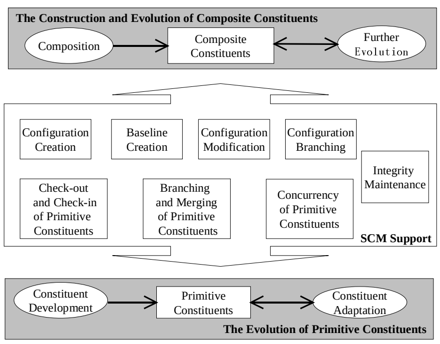

Figure 1 Concept Model of JBCM

The concept model of JBCM is depicted in Figure 1. At bottom of the figure is the evolution of primitive constituents. On top of the figure is the construction and evolution of composite constituents. Both kinds of constituents should be under the management of JBCM. In this model, there are some operations for the management of both primitive and composite constituents.

3.2. Management of Primitive Constituents

In this model, a primitive constituent that is made of a group of files rather than an individual file is treated as the basic unit of version control, and the operations for management of primitive constituents are check-out, check-in, branching, merging, and concurrency control.

- **Check-out and check-in.** The check-out-edit-check-in style is used to manage each evolution direction of a primitive constituent. To make a change to a primitive constituent, it should be checked out first. After finishing the change, it should be checked in and thus add a new version.
- **Branching and merging.** A primitive constituent can have several evolution directions. By using the branching operation, a new evolution direction can be created from any of the existing versions of a primitive constituent. An evolution direction is also called a branch. Different branches can be merged together during the evolution of the primitive constituent.
- **Concurrency.** A primitive constituent is usually under the development of a team. The concurrency operation is used to coordinate the different modifications from different developers to files in a primitive constituent.

3.3. Management of Composite Constituents

In this model, a composite constituent is represented by a configuration, thus the operations for the management of composite constituents can be represented by the operations on configurations, which are configuration creation, baseline creation, configuration modification, and configuration branching.

- **Configuration creation.** By selecting some primitive constituents and some existing configurations, a new configuration can be created. A configuration that serves as an element of another configuration is also called a sub-configuration.
- **Baseline creation.** A composite constituent will evolve with the evolution of its constituents. Selecting a version for each primitive constituent in the configuration, and selecting a baseline for each sub-configuration in the configuration, a new baseline for the configuration can be created. This operation is called baseline creation. A baseline represents a milestone version of a composite constituent.
- **Configuration modification.** How a composite constituent is composed can also be changed. Modification to a configuration represents this kind of change.
- **Configuration branching.** A composite constituent can also have several evolution directions. The branching operation of a configuration can be used to manage all the evolution directions of the corresponding composite constituent.

3.4. Maintenance of integrity

In this model, operations for maintenance of integrity of constituents include the following two aspects.

- **Consistency and integrity checking.** It is useful to check if there is any inconsistency between primitive constituents and/or composite constituents, while the composed system is under frequent changes.
- **Change affection analysis.** When a change is made to a composite constituent or a primitive constituent, it is needed to analyze the affection of the change to help to make the corresponding changes of other primitive constituents and/or composite constituents.

4. Design and Implementation of JBCM

In this section, we first present the architecture of JBCM, which is divided into three layers: the mechanism layer, the basic function layer, and the high-level function layer; and then we present implementation details of each layer.

4.1. Architecture

The architecture of JBCM is depicted as Figure 2, which conforms to the federated CM architecture proposed in [3].

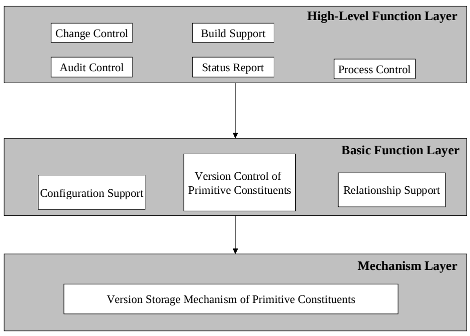

Figure 2 Architecture of JBCM

- **Mechanism layer.** The main tasks of the mechanism layer are to implement primitive constituent versions based on file versions and to store versions incrementally.
- **Basic function layer.** The basic function layer is the kernel of the system. It covers the features discussed in the concept model. This layer is based on the mechanism

layer, and all the functions in this layer are implemented via the mechanism layer.

- **High-level function layer.** The functions in this layer are not mentioned in the concept model. However, as these functions have all been addressed in literature, such as [7] and [5] etc., we think they are necessary in a real world CM system.

4.2. Mechanism layer

The mechanism layer is transparent to users. It mainly fulfils the following tasks:

- **Implementing primitive constituent versions based on file versions**

In JBCM, we use an auxiliary file to record the ingredients of a primitive constituent. A version of the primitive constituent is represented by a version of the auxiliary file, which denotes the corresponding versions of files that the primitive constituent is made of. Therefore, the operations on a version of the primitive constituent can be transparently mapped to the operations on a group of file versions via the auxiliary file.

- **Incremental storage of primitive constituent versions**

In JBCM, versions of both the auxiliary file and the ingredient files of the primitive constituent are stored incrementally using the reverse incremental storage technique investigated in RCS [15]. The difference between JBCM and RCS on incremental storage is, that JBCM stores each branch individually and uses the reverse incremental storage technique in storing versions on each branch while RCS stores versions on all the branches together and uses the reverse incremental storage on only one branch.

4.3. Basic Function Layer

The basic function layer provides three functions: versioning of primitive constituents, configuration support, and relationship support.

- **Version control of primitive constituents**

In JBCM, it is allowed for a team of developers to concurrently modify one primitive constituent. Based on the mechanism described above, we use operation modes of files to control concurrent modification. There are three modes: "*read-only*", "*exclusive-write*", and "*share-write*". Only when a developer has a file in *exclusive-write* mode or *share-write* mode, can he/she make changes to the file. Using *share-write* mode allows more concurrency, but it relies on the automatic merging of two file versions, and the correctness of this kind of merging cannot be guaranteed.

The procedure of the modification to a primitive constituent is as follows. Before a developer makes the modification to the constituent, he/she should check out the constituent first with originally the operation modes of all files set to *read-only*. Then before he/she makes a change to a file, he/she should change the operation mode of the file from *read-only* to *exclusive-write* or *share-write*. If the file is currently under modification of another developer, he/she cannot change the operation mode, and thus cannot make any change to the file. After changing the mode, he/she can make the actually change. After all the changes to the files are done, he/she can check in the primitive constituent to add a new version of the constituent.

To a primitive constituent, this procedure still conforms to the conventional style of *check-out-edit-check-in*, but to a file, it is explicit for the developer to change the operation modes of files.

- **Configuration support**

In JBCM, we use a file to implement a configuration and its baselines. Both the definition of the configuration and the definition of its baselines are represented as records in the file.

- **Relationship support**

As the techniques for consistency checking and change affection analysis still need further investigation, we only implement a basic facility for change affection analysis. In JBCM, users can define relationships between constituents, and therefore the analysis of the affection of a change to a constituent can be performed based on previously defined relationships.

4.4. High-level function layer

The main problem of implementing the high level functions is that they are built on version control of files in the other systems, but they will be built on version control of primitive constituents in JBCM. However, as we have a transparent mechanism layer to map operations on primitive constituents to operations on files, it is almost the same to implement the functions in our system as in other configuration management systems. Among these functions, although we only have implemented part of build support and status support by now, we believe that all the functions can be implemented in JBCM in the future.

5. An Example

5.1. A simplified domestic temperature control system

To illustrate how our system works, we give a simple example here. Figure 3 depicts a simplified domestic temperature control system in winter. The clock gives a signal to the controller periodically. When receiving a signal, the controller reads the temperature of the home from the thermometer. If the temperature is higher than the high limit, the controller will turn off the heater. If the temperature is lower than the low limit, the controller will turn on the heater.

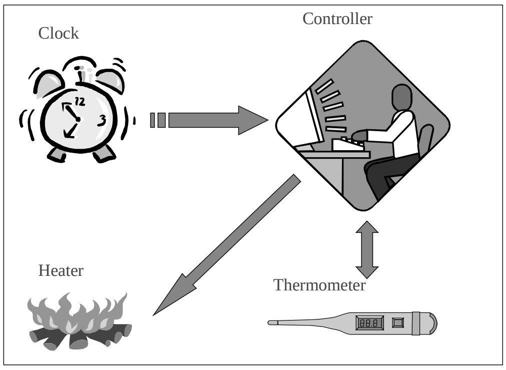

Figure 3 Domestic Temperature Control System

5.2. A possible development process

To build such a system, we may have the following development process: 1) that we select four corresponding prefabricated primitive components, each of which is implemented by a class in C++; 2) that we find the thermometer cannot interact with the controller properly, so we modify the interface and implementation of the thermometer, and get a system that is roughly acceptable; and 3) that we also find that clock does not function as we expect, so we modify its implementation.

Considering that we want to use the system for both heating in the winter and cooling in the summer, we will add a cooler into the system to extend its ability. We may have the following process: 1) that we select a prefabricated cooler component, which is implemented in C++; 2) that to incorporate the new component, we modify the controller to make it be able to control the cooler and thus get a fairly good system; and 3) that we further make some modifications to the heater and the cooler to make them work in similar pace.

5.3. Applying the system

Suppose the above-mentioned five components are implemented by the following ten files in C++: "clock.h", "clock.cpp", "controller.h", "controller.cpp", "thermometer.h", "thermometer.cpp", "heater.h", "heater.cpp", “cooler.h” and “cooler.cpp”. To support the above development process, we will use the following operations in the system.

To represent the original system, we create a configuration. To adapt the component thermometer, we check out the component, modify both the file “thermometer.h” and the file “thermometer.cpp”, check it in, and add a new version. To record this version, we create a baseline for the configuration. To adapt the component clock, we check out the component, modify the file “clock.cpp” in it, check it in, and add a new version. We therefore created a new baseline for the configuration.

To represent the new system, we create a new configuration. To adapt the component controller, we check out the component, modify both the file “controller.h” and the file “controller.cpp”, check it in, and add a new version. To record this version, we create a baseline for the configuration. To adapt the component heater and the component cooler, we check out the components respectively, modify the files “heater.cpp” and “cooler.cpp” in corresponding components, check them in respectively and add two new versions. We therefore created a new baseline for the configuration.

The two configurations and five primitive components are depicted in Figure 4. For each component, the version 1.1 is the original version, and the version 1.2 is created through the operation of check-out-edit-check-in of the component. For each configuration, there are also two corresponding baselines, which are not shown in the figure. The first baseline for the original system is made of version 1.1 of the clock, version 1.1 of the controller, version 1.2 of the thermometer, and version 1.1 of the heater. The second baseline for the original system is made of version 1.2 of the clock, version 1.1 of the controller, version 1.2 of the thermometer, and version 1.1 of the heater. The first baseline for the new system is made of version 1.2 of the clock, version 1.2 of the controller, version 1.2 of the thermometer, version 1.1 of the heater and version 1.1 of the cooler. The second baseline for the new system is made of version 1.2 of the clock, version 1.2 of the controller, version 1.2 of the thermometer, version 1.2 of the heater and version 1.2 of the cooler.

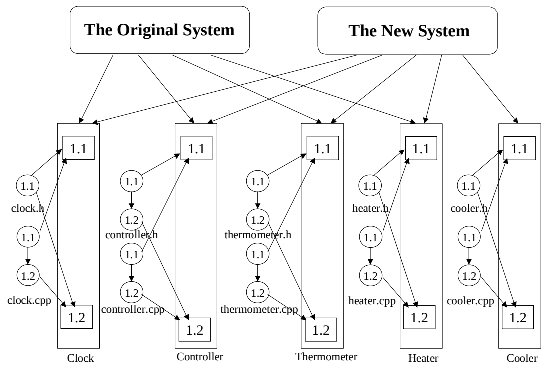

Figure 4 Primitive Components and Configurations

5.4. A comparison with other CM systems

If a previous CM system such as RCS is applied to support the development of the system in this example, the files and the configurations should be like what is depicted in Figure 5.

From Figure 5, we can see 1) that the versions of any component are not recorded; and 2) that components are difficult to identify and thus such a CM system cannot be helpful enough in maintaining the integrity between components.

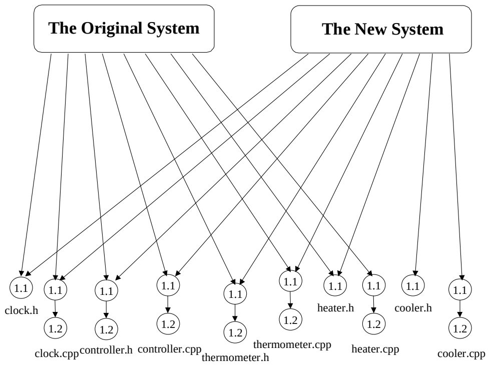

Figure 5 Files and Configurations in a Previous CM System

6. Conclusion

In this paper, we have summarized the requirements of CM for CBSD, and presented a prototype system to meet the requirements. In this prototype system, we treat each primitive constituent rather than each individual file as the basic unit; we exploit recursively-defined configurations to represent composite constituent; and we use user-defined relationships to assist change affection analysis. We believe that the system can support CBSD more efficiently than previous CM systems. Future directions of research include finding more effective techniques for integrity maintenance in CBSD, techniques for constituent merging at syntax level, and techniques for logical versioning of constituents.

References

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

[2] E. Borison, A Model of Software Manufacture, in Advanced Programming Environments, R. Conradi, T. Didriksen and D. Wanvik, Eds. New York: Springer-Verlag, 1986, pp. 197-200.

[3] A. Brown, S. Dart, P. Feiler, and K. Wallnau, The State of Automated Configuration Management. Technical Report Annual Technical Review, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 1992.

[4] A. W. Brown, and K. C. Wallnau, Engineering of Component-Based Systems, Component-Based Software Engineering: Selected Papers from the Software Engineering Institute. Los Alamitos, CA: IEEE Computer Society Press, pp. 7-15. 1996.

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

[6] 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.

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

[8] Luqi, A Graph Model for Software Evolution, IEEE transactions on Software Engineering, vol. 16, no. 8, pp. 917-927, Aug. 1990.

[9] 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.

[10] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering, Vol. 21, No. 6, pp528-561, June. 1995.

[11] M. Moriconi, and T.C. Winkler, Approximate reasoning about the semantic effects of program changes, IEEE transactions on Software Engineering, vol. 16, no. 9, pp. 980-992, Sept. 1990.

[12] K. Narayanaswamy and W. Scacchi, Maintaining Configurations of Evolving Software Systems, IEEE transactions on Software Engineering, vol.13, no. 3, pp. 324-334. March. 1987.

[13] E. Ploedereder and A. Fergany, A Configuration Management Assistant, In Proceedings of the Second International Workshop on Software Version and Configuration Control, pages 5-14. ACM, USA, October 1989.

[14] A. Podgurski, and L.A. Clarke, A Formal Model of Program Dependences and its Implications for Software Testing, Debugging and Maintenance, IEEE transactions on Software Engineering, vol. 16, no. 9, pp. 965-979, Sept. 1990.

[15] W. Tichy, RCS – A System for Version Control, Software Practice and Experience 15, 7, 637-654. 1985.

[16] 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.

[17] F. Yang, H. Mei, and K. Li. Software Reuse and Software Component Technology, Acta Electronica Sinica, vol. 27, no. 2, pp. 68-75, Feb. 1999. (In Chinese)