Towards Component-based Software Maintenance via Software Configuration Management Techniques

Leonardo Murta, Hamilton Oliveira, Cristine Dantas,
Luiz Gustavo Lopes, Cláudia Werner
{murta, hamilton, cristine, luizgus, werner}@cos.ufrj.br
COPPE/UFRJ – System Engineering and Computer Science Program
Federal University of Rio de Janeiro – P.O. Box 68511
ZIP 21945-970 – Rio de Janeiro – RJ – Brazil

Abstract

This paper presents an approach that aims to assist maintenance of component-based systems by means of Software Configuration Management techniques. These techniques support different activities of software maintenance, from the maintenance request up to implementation and integration. Moreover, some feedback about the relationship of component-based artifacts is provided by applying data mining techniques over configuration management repositories.

1. Introduction

The software development process encompasses many distinct phases, each of them related to a specific abstraction level. This multi-level structure is important to model and refine the knowledge about a domain, providing control over the complexity of software development. At the early phases, knowledge, represented by analysis artifacts, is tightly related to the problem description. These artifacts are gradually refined and used as source of information to build other artifacts, which aim to describe a solution for the problem at hand.

When analyzing this process, it can be noticed that some semantic structures might be repeated in different phases. The same concept can be represented from different points of view, depending on the goal of the phase. In the beginning of the process, the main objective is to describe the problem. For this reason, the requirements are elicited and the key concepts are identified. In the late phases of the process, the same concepts will be detailed in a deeper way, trying to describe how they can be represented by computer programs.

Some decades ago, completely different notations were used to construct artifacts of these different phases of the development process. However, in the last decades some initiatives have tried to shorten the gap between the problem definition and the solution definition. The most known initiatives are Object-Oriented development, Aspect-Oriented development and Component-Based Software Engineering (CBSE).

Object-Oriented development aims to use the concept of class from analysis up to coding phases. Different modeling notations and programming languages were created to support the concept of class and other related concepts. In the same way, Aspect-Oriented development is being used as a complement to Object-Oriented development in representing crosscutting features of systems, such as non-functional requirements.

However, the concept of class is too fine grained to be reused or replaced in a systematic way. On the other hand, CBSE provides a more cohesive and coarse grained

structure. These characteristics are well suited to describe parts of a system, helping to reuse or changing them. Nevertheless, to increase the productivity and quality, a complete infrastructure is needed to support maintenance of component-based systems during execution of complex and error-prone related activities.

During the software maintenance phase, traces between artifacts in different abstraction levels that represent the same concept should be preserved and evolved consistently. In the case of CBSE, for example, a very precise impact analysis should be performed to avoid incompatibilities among different versions of the same component.

Software Configuration Management (SCM) systems can be used to help solving this problem. However, the current systems are not suited to the coarse grained artifacts used in CBSE [17, 18]. This paper presents an approach for component-based maintenance integrating SCM functionalities to the CBSE environment.

The remaining of this paper is organized in four sections. Section 2 introduces SCM as a discipline that can be applied to software maintenance. Section 3 discusses the proposed approach for component-based maintenance via SCM, contrasting them with some related work. Finally, Section 4 concludes this paper with a discussion about contributions, limitations, and future work.

2. Software Configuration Management

Software Configuration Management is a discipline for controlling the evolution of software systems. SCM is applicable throughout the product's lifetime – from creation, development, product release, customer delivery, customer use, up to maintenance. Ideally, SCM systems should support this with low overhead, thereby allowing SCM systems to be applied as early as possible on a project [7].

A definition taken from IEEE STD 729 [11] highlights the following aspects:

- Identification: reflects the structure of the product, by identifying the configuration items and their types, and making them unique and accessible in some form.
- Control: controlling the release of a product and changes to it throughout the lifecycle by having mechanisms in place that ensure consistent software via the creation of a baseline product.
- Status accounting: recording and reporting the status of configuration items and change requests, and gathering vital statistics about them in the product.
- Audit and review: validating the completeness of a product and maintaining consistency among the configuration items.

SCM techniques are commonly used to control the evolution of software systems by providing version control and support for activities related to change management. With SCM systems, software engineers efficiently build a consistent software product and communicate and coordinate by notifying one another about required and completed tasks.

3. Component-based maintenance via SCM

In a complex scenario of components evolution, components are usually adapted before reused. Although SCM supports the needs of conventional software development, it has been recognized that it does not meet the specific demands of CBSE [22]. Since the current systems are not suited to the high-level artifacts used by CBSE, it is necessary to adapt them to fit this need. This new kind of SCM systems should help in managing a

consistent evolution of independently developed, but interrelated, set of components, integrating traditional SCM functionality within CBSE activities.

This paper focuses on the application of SCM techniques within a CBSE environment providing control over reused artifacts and support to component-based maintenance activities [6]. In the next sections, we discuss three topics of this approach: automation of maintenance processes, control over the evolution of CBSE artifacts, and detection of change traces among these artifacts.

3.1. Automation of maintenance processes

CBSE differs from conventional software engineering approaches in several ways. First, the life cycle in CBSE has different activities, such as finding components, selecting those that appropriately fit the requirements, adapting them when necessary, deploying components in a component infra-structure and replacing the ones with defect [16].

In addition, there is a difference concerning the players related to these approaches [16]. In conventional software engineering, there are engineering teams and end-users, while in CBSE there are component producer teams, which develop components, component consumer teams, which develop software reusing components, hybrid teams, which are both consumer and producer, and end-users.

Therefore, changes over components demand some adaptations to SCM processes to fit in this new scenario. However, these processes are still immature, and the main standards defined up to now for SCM [12, 13, 14] do not cover the specific CBSE issues. Then, change control systems (CCS) should be flexible enough to support process customization, even for running processes. By providing this feature, it is possible to use experience to improve processes, even at runtime, without crashing them or losing information.

Moreover, new information needs to be collected in SCM processes for CBSE, such as who are the component producers, where and how to locate them. However, there is no agreement about which kind of information to collect. Therefore, a CCS must allow the definition of which information will be collected in SCM processes for CBSE.

In order to solve these problems, we propose a CCS specifically designed for this scenario, which provides process and information customization. The solution defines compound and primitive processes. A compound process is associated to a process definition, which describes its sub processes and their relationships. For instance, the change control process itself would be modeled as a compound process. Primitive processes represent SCM activities, and enable end-user interaction with the system. For the process definition, we will use some approach such as SPEM [19] or IDEF3 [10].

While modeling a process, the configuration manager associates primitive processes with forms, which can be customized by defining which information should be requested, and which field type should be used to collect it. Currently, the modeled forms used in process enactment are created in HTML [23], but we are evaluating other approaches, such as using the Web form specification XForms [24], the framework CForms [1] and the template engine Velocity [2].

Each process complete cycle generates a new change object, which aggregates a set of documents containing the information provided by end-users. Therefore, they provide all information collected in SCM activities. These changes may be grouped in a change set,

which makes it easier to exchange information among component producer and consumer teams.

There are some other issues that need to be addressed, concerning the proposed solution, such as how to keep traces among component producers and consumers, to allow a component consumer to contact the producers and vice-versa, enabling change request forwarding when required. Another issue to be explored is the need for a notification mechanism, to allow people to be informed about process events.

Some approaches that consider change control features for the CBSE context are TERRA [15] and KobrA [3]. However, TERRA provides a fixed SCM process, which may only be adapted by source code modifications. In addition, it does not provide form customization capability, being difficult to adapt the system to different request information. The KobrA method brings interesting features, such as change propagation based on change sets and cause relationship between changes.

3.2. Controlled evolution of CBSE artifacts

Version control system is the subsystem that received more attention of the market and of research on SCM. Current version control systems use a simple data model based on file system. This data model is not appropriate to support modeling environments that use complex data structures, such as component.

Two types of artifacts can be identified in SCM: *basic and composite configuration items* [20]. A composite configuration item is a collection of basic configuration items and other composite configuration items. A component is essentially a composite configuration item that entails other configuration items, such as class, use cases, packages, source and binary code. It expresses a composition relationship.

On the other hand, when a component uses the services provided by another component, we have a dependency relationship. Figure 1 shows Component B v2.3 using the services provided by Component A v5.9, constituting a dependency relationship. Inside the components there are other elements, constituting a composition relationship. Any modification to elements contained in Component A should increase its version. This new version can motivate a new version of Component B.

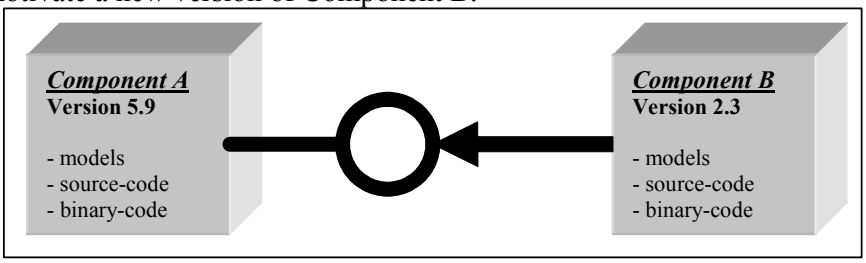

Figure 1. Composition and dependency relationship in CBSE.

The information at component level is described through model elements. Among these artifacts, only the source-code can have its evolution controlled by current version control system in a satisfactory way. The problem is the coarse granularity, consequence of the use of a simple data model based on file system. In these systems, a file is treated as a single entity, therefore a model (either Component model or UML model) persisted in a file is versioned as an indivisible item. In this approach it is not possible to get all the versions of a business type, package, class, or use case.

Christensen [4] and Atkinson et al. [3] propose version control systems to CBSE. Dependency relationship, connectors, interfaces; architectural elements are taken into account by these systems. However, these systems do not control the evolution of model elements individually.

In our approach it is possible to version model elements, individually. The model element should be defined as a *Unit of Version* (UV). Therefore, these elements will be considered configuration items when sent to the repository. Actually, our tool is able to version any model element in UML. This approach can be extended to version Components model, through the definition of UML Stereotypes that allow to represent business type like a class, or through an upgrade of the metamodel UML 1.4 to metamodel UML 2.0. UML 2.0 permits to represent components model.

However, treating each model element individually introduces new problems that do not exist in the other approaches. For example, a business type and a package have different characteristics. To solve this problem a generic versioning interface is used. Each element that is defined as UV should have a specific *Handler* class that is responsible for its evolution.

3.3. Detection of change traces among CBSE artifacts

During the maintenance phase, developers must ensure that related software artifacts are updated to be consistent as the software system evolves. Traceability techniques are used to identify all artifacts that should be updated when a change is introduced [5].

Software artifact traceability is widely recognized as an important factor for effectively managing the development and evolution of software systems. Traceability is also fundamental to help software comprehension, maintenance, impact analysis and reuse of existing software elements.

Currently, time-pressured practitioners fail to consistently maintain traces and update impacted artifacts each time a change occurs. The activity of manual detection and maintenance of traces, which has to be done frequently due to the interactive nature of software development, is time-consuming, error prone, and person-power intensive. One of the main problems of maintenance is the lack of automatic or semi-automatic trace generation and maintenance tools [5].

As CBSE uses components, interfaces, and connectors as first-class elements for structuring systems, the maintenance problem can be minimized with an approach to gradually support the identification of traces between high-level artifacts that are related through the component concept.

Recently, researches have experienced the use of SCM repositories to understand software as well as their evolution. Change data have been used by various researchers for qualitative and quantitative analysis. Gall et al. [8] were the first to use release data to detect logical coupling between modules. Sayyad-Shirabad et al. [21] use inductive learning to learn different concepts of relevance among logically coupled files. Several research tools that support traceability among program files [21] and among fine-grained program entities [25] are available. However, these tools are focused on source code, not paying much attention to analysis and design artifacts.

For example, when a software engineer changes a *UML* model, one of the main questions he or she needs to answer is “Are there other elements that may be affected by

these changes?" One way to assist software engineers to answer this question is finding the relations of existing *UML* model elements, which indicates the elements that will need to be changed together. Therefore, two types of relationships are highlighted: among the same conceptual element in different levels of abstraction and among different conceptual elements, however, in the same abstraction level.

In our approach, these relationships are discovered by mining over software configuration management repositories. A considerable amount of system maintenance experience can be found within the data of SCM systems. By applying data mining techniques over the version control system described in Section 3.2, we generate change traces among *UML* elements and also guide developers along related changes. By analyzing semantic relations among artifacts that are related through common changes, the software engineer can identify the potential side-effects of a change, estimating what is needed to be modified to accomplish that change. Consequently, if the user changes some artifact, this approach automatically recommends possible future changes.

As developers investigate change traces, they pose various questions to uncover the rationale for the dependencies that were identified. Currently, the tools that support traceability [25] are not appropriate due to the lack of semantic for the relations among software artifacts. To resolve this problem, our tool classifies these questions into six categories: who, when, how, why, what and where. In the world of software engineering with tight schedules and short time to market, manually recording such information is neither possible nor practical. On the other hand, SCM repositories store change details collected in SCM activities described in Section 3.1. This approach, as illustrates Figure 1, provides a view of all related artifacts that should be updated with details about the rationale, the history or the people behind the dependency relations. Such details are vital in assisting developers through the modification process [9].

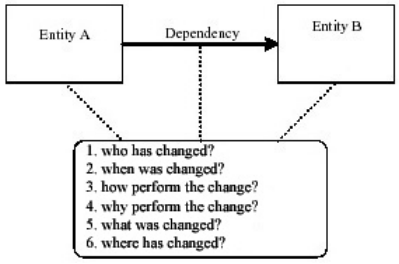

Figure 2. Rationale for change traces.

In the task of maintaining software artifacts, this approach is not intended to replace the work of the software engineer. Since change trace is based on experience, it does not constitute absolute truth, but a suggestion. To give an idea of how much the suggestion is recommended, each change trace has a probabilistic interpretation based on the amount of evidence from the data they are derived from.

Since software artifact traceability is fundamental to effectively manage the development and evolution of software systems, this approach aims to help software engineers to suggest and predict likely further changes within a CBSE environment.

4. Conclusions

This paper presented an approach for component-based maintenance via SCM techniques. The discussed aspects were maintenance processes automation; CBSE artifacts evolution and change trace detection. Currently, a process machine tool for CBSE is under implementation, and both the artifacts evolution and traces tools have a first versions already implemented. Experimental studies verifying this approach is a concern to be considered in future researches.

The main contribution of this approach is related to the integration of different SCM techniques to help CBSE maintenance. Other generic approaches do not support specific characteristics of CBSE, or focus on only very narrow aspects, not spreading the benefits of a controlled environment over the different maintenance phases.

The maintenance approach presented here is part of a wider Project that aims to provide SCM support for both development and maintenance phases of CBSE. Besides maintenance processes automation, CBSE artifacts evolution and change trace detection, other aspects of software maintenance, such as the SCM process itself and build and release activities, are topics of interest of our research group.

Acknowledgements

The authors would like to thank CAPES, CNPq and Banco Central do Brasil for their financial support.

References

[1]Apache, 2004a, Cocoon Forms: Introduction In: http://cocoon.apache.org/ 2.1/ userdocs/forms/index.html, Accessed in 08/2004.

[2]Apache, 2004b, The Apache Jakarta Project - Velocity, in: http://jakarta.apache.org/ velocity/, Accessed in 08/2004.

[3]Atkinson, C., Bayer, J., Bunse, C., et al., 2001, Component-Based Product Line Engineering with UML. 1st. ed., Addison-Wesley.

[4]Christensen, H. B., 1999b, The Ragnarok Architectural Software Configuration Management Model. In 32nd Annual Hawaii International Conference on Systems Sciences, Maui, Hawaii, January.

[5]Cleland-Huang, J., Chang, C. K., 2003, Event-Based Traceability for Managing Evolutionary Change in IEEE Transactions on Software Engineering, v.29, n.9, p: 796-810, September.

[6]Dantas, C. R., Oliveira, H. L. R., Murta, L. G. P., et al., 2003, Um Estudo sobre Gerência de Configuração de Software aplicada ao Desenvolvimento Baseado em Componentes In: Terceiro Workshop de Desenvolvimento Baseado em Componentes (WDBC), São Carlos, SP, September.

[7]Dart, S., 1991, Concepts in Configuration Management Systems, Third International Software Configuration Management Workshop, ACM Press, p: 1-18, Trondheim, Norway, June.

[8]Gall, H., Jazayeri, M., Klösch, R., Trausmuth, G., 1997, Software Evolution Observations based on Product Release History in Proceedings of International Conference on Software Maintenance (ICSM 1997), Bari, Italy, p: 160-196.

[9]Hassan, A. E., Holt, R. C., 2004, Using development history sticky notes to understand software architecture in Proceedings 12th IEEE International Workshop on Program Comprehension(IWPC'04), p:183-192, Bari, Italy, June.
[10]IDEF, 2004, IDEF3 Process Flow and Object State Description Capture Method Overview, in: http://www.idef.com/idef3.html, Accessed in 08/2004.
[11]IEEE 729-1983, 1982, Glossary of Software Engineering Terminology, September.
[12]IEEE 828-1998, 1998, IEEE Standard for Software Configuration Management Plans, May.
[13]IEEE 1042-1997, 1997, IEEE Guide to Software Configuration Management, September.
[14]ISO 10007, 1995, Quality Management - Guidelines for Configuration Management, April.
[15]Kwon, O., Shin, G., Boldyreff, C., et al., 1999, Maintenance with Reuse: An Integrated Approach Based on Software Configuration Management. In: Sixth Asia Pacific Software Engineering Conference, p: 507-515, Takamatsu, Japan, December.
[16]Larsson, M., 2000, Applying Configuration Management Techniques to Component-Based Systems, Licentiate Thesis, Department of Information Technology, Uppsala University, Sweden.
[17]Murta, L. G. P., 2004, Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes, Exame de Qualificação. COPPE/UFRJ, Rio de Janeiro, RJ, Brazil, May.
[18]Oliveira, H. L. R., Murta, L. G. P., 2004, Odyssey-VCS: Um Sistema de Controle de Versões Para Modelos Baseados no MOF In: XVIII Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas, Brasília, DF, Brazil, October.
[19]OMG, 2004, Software Process Engineering Metamodel, in: http://www.omg.org/technology/documents/formal/spem.htm, Accessed in 08/2004.
[20]Pressman, R. S., 2000, Software Engineering – A Practioners Approach, 5a ed., McGraw Hill.
[21]Sayyad-Shirabad, J., Lethbridge, T. C., Matwin, S., 2001, Supporting Software Maintenance by Mining Software Update Records in Proceedings of the 17th IEEE International Conference on Software Maintenance, p: 22-31, Florence, Italy.
[22]Sowrirajan, S., Hoek, A., 2003, Managing the Evolution of Distributed and Interrelated Components in SCM-11, p: 217-230, Portland, Oregon.
[23]W3C, 1999, HTML 4.01 Specification. World Wide Web Consortium.
[24]W3C, 2004, XForms: The Next Generation of Web Forms, in: http://www.w3.org/MarkUp/Forms/, Accessed in 08/2004.
[25]Zimmermann, T., Weisgerber, P., Diehl, S.; Zeller, A., 2003, Mining version histories to guide software changes in Proceeding International Conference on Software Engineering (ICSE 2004), EICC, p: 563-572, Scotland, UK, May.