Software Configuration Management

E. James Whitehead, Jr. Annita Persson Dahlqvist (Eds.)

Software Configuration Management

Proceedings of the 12th International Workshop on Software Configuration Management (SCM 2005)
Lisbon, Portugal, September 5-6, 2005
Held in conjunction with ESEC/FSE 2005

Editors

E. James Whitehead, Jr.
Dept. of Computer Science
University of California, Santa Cruz
1156 High Street, SOE3
Santa Cruz, CA 95064
ejw@cs.ucsc.edu

Annita Persson Dahlqvist
Ericsson AB
Mölndal, Sweden
annita.persson.dahlqvist@ericsson.com

12th International Workshop on Software Configuration Management
http://purl.oclc.org/NET/SoftwareConfigurationManagement/SCM2005/

CR Subject Classification (1998): K.6.3, K.6, D.2

ISSN: 0302-9743

Preface

In our complex technological civilization, the multidisciplinary teams who develop large engineering projects produce vast quantities of documents representing multiple views and perspectives of the developing system. These documents—such as requirements, blueprints, designs, email, source code, test plans, and standards—are highly interrelated, and taken together form a complex information artifact that describes the system. Systems are constantly modified in response to changes in the environment or marketplace, a process that involves changing the associated information artifact. So that system modifications proceed from full knowledge of the system's current state, all engineering activities share a common need to record the configuration of the documents that describe the system. Configuration management facilities that track the evolution of complex information artifacts over time are necessary for tracking system configurations.

Today, software is a component of nearly all systems we construct. Due to software's malleability, and the large impact that even a small modification can make, Software Configuration Management (SCM) has long been important in Software Engineering. Indeed, it was in 1975, as the typical programmer experience was shifting from batch work using punched cards and tape reels—capable of configuration control using paper-based techniques—to timeshared development of code stored on hard disks, that the first developer-oriented SCM system, SCCS, was developed. The intervening three decades of research and commercial development have witnessed the creation of scores of commercial and open source SCM systems. A testament to the continuing relevance of SCM is that commercial sales of SCM tools comprise a billion dollar a year marketplace, highlighting that SCM tools solve coordination and control problems so severe that organizations are willing to make significant investment to solve them. SCM systems are part of the everyday experience of today's software engineers, and hence research that leads to improved understanding and capabilities of SCM systems can have broad impact on software engineering practice.

This volume contains the research papers presented at the Twelfth International Workshop on Software Configuration Management (SCM 2005), held September 5-6, 2005, in Lisbon, Portugal. The workshop was held as part of the 2005 European Software Engineering/Foundations of Software Engineering (ESEC/FSE 2005) conference, and was attended by 22 people. The workshop received 16 paper submissions, 10 of which were accepted. We would like to thank the workshop's Program Committee for their diligence in providing high quality reviews of all submissions.

August, 2005

Jim Whitehead
Annita Persson Dahlqvist

SCM Workshops

1. International Workshop on Software Version and Configuration Control (SVCC), Grassau, Germany, January, 1988
2. Second International Workshop on Software Configuration Management (SCM-2), Princeton, New Jersey, October 24, 1989 (ACM Press, Software Engineering Notes, Vol. 17, No. 7, Nov. 1989)
3. Third International Workshop on Software Configuration Management (SCM-3), Trondheim, Norway, June 12-14, 1991 (ACM Press)
4. Fourth International Workshop on Software Configuration Management (SCM-4), Baltimore, Maryland, 1993 (Springer, Lecture Notes in Computer Science (LNCS) 1005)
5. Fifth International Workshop on Software Configuration Management (SCM-5), Seattle, Washington, May, 1995 (Springer, LNCS 1005)
6. Sixth International Workshop on Software Configuration Management (SCM-6), Berlin, Germany, March, 1996 (Springer, LNCS 1167)
7. Seventh International Workshop on Software Configuration Management (SCM-7), Boston, Massachusetts, May, 1997, (Springer, LNCS 1235)
8. Eighth International Symposium on System Configuration Management (SCM-8), Brussels, Belgium, July, 1998 (Springer, LNCS 1439)
9. Ninth International Symposium on System Configuration Management (SCM-9), Toulouse, France, September, 1999 (Springer, LNCS 1675)
10. Tenth International Workshop on Software Configuration Mangement (SCM-10/SCM 2001), Toronto, Canada, May, 2001 (Springer, LNCS 2649)
11. Eleventh International Workshop on Software Configuration Management (SCM-11/SCM 2003), Portland, Oregon, May, 2003 (Springer, LNCS 2649)
12. Twelfth International Workshop on Software Configuration Management (SCM 2005), Lisbon, Portugal, September, 2005 (ACM Digital Library)

Proceedings for past SCM workshops can be found either in the ACM Digital Library (1989, 1991, 2005), or in Springer Link, the Springer digital library (all others except 1988). The proceedings of the first SCM workshop (SVCC) are not available online, and are generally difficult to find even in print form.

Program Committee of SCM 2005

Program Chairs

- Jim Whitehead, University of California, Santa Cruz, USA
- Annita Persson Dahlqvist, Ericsson AB, Mölndal, Sweden

Committee Members

- Geoff Clemm, IBM Rational, USA
- Reidar Conradi, NTNU Trondheim, Norway
- Ivica Crnkovic, Malardalen University, Sweden
- Wolfgang Emmerich, University College London, United Kingdom
- Jacky Estublier, Centre National de la Recherche Scientifique, France
- André van der Hoek, University of California, Irvine, USA
- René L. Krikhaar, Philips Medical Systems, Netherlands
- Bernhard Westfechtel, University of Bayreuth, Germany
- Andreas Zeller, University of Saarbrücken, Germany

Table of Contents

Odyssey-VCS: a Flexible Version Control System for UML Model Elements

Hamilton Oliveira, Leonardo Murta, Cláudia Werner

COPPE/UFRJ – Systems Engineering and Computer Science Program
Federal University of Rio de Janeiro – P.O. Box 68511
21945-970 Rio de Janeiro, Brazil
{hamilton, murta, werner}@cos.ufrj.br

Abstract. Many current version control systems use a simple data model that is barely sufficient to manipulate source-code. This simple data model is not sufficient to provide versioning capabilities for software modeling environments, which are strongly focused on analysis and architectural design artifacts. In this work, we introduce a flexible version control system for UML model elements. This version control system, named Odyssey-VCS, deals with the complex data model used by UML-based CASE tools. Moreover, it allows the configuration of both the unit of versioning and unit of comparison for each specific project, respecting the different needs of the diverse development scenarios.

1 Introduction

Computer Aided Software Engineering (CASE) tools can be classified into two main groups [24]: lower CASE tools and upper CASE tools. Lower CASE tools are mostly concerned about implementation and testing issues; whereas upper CASE tools deal with higher abstraction levels, entailing requirements, analysis, and design artifacts. Besides the necessity of Software Configuration Management (SCM) support for every kind of CASE tool [21], this support has been typically focused on lower CASE tools, providing a huge infrastructure over the last decades to leverage evolution of source-code artifacts.

However, due to the increasing software development complexity, SCM support is also needed by upper CASE tools. Model-driven development is emerging as a promising technique for complexity control. Model-driven approaches focus on the definition of high level models and apply subsequent transformations to obtain implementation artifacts. Nevertheless, the current SCM infrastructure does not properly support the evolution of model-based artifacts.

A first thought would be to adapt the existing SCM techniques, formerly applied to source-code, to this new context. However, current SCM infrastructures are not suited to the coarse grained artifacts used by upper CASE tools. For example, most current SCM systems are based on file system structures, while upper CASE tools are based on higher level structures. The mapping of these complex structures used by upper CASE tools to file structures is dangerous due to concept mismatch.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

Moreover, most SCM standards [10, 11] recommend the selective identification of Configuration Items (CI) that depend on individual characteristics of the software development projects. Nearly all state-of-the-practice SCM systems have a fixed identification of CI: the file. Due to this fact, every artifact that needs versioning information should be stored into an individual file. However, in some circumstances it is neither desirable nor possible to map every high level analysis and design artifact into an individual file.

Aiming to diminish the effects of these problems, we propose a novel approach to support UML-based upper CASE tools in evolving their artifacts. This approach, named Odyssey-VCS, consists of a Version Control System (VCS) for UML model elements that can be tailored to the specific needs of each software development project, as recommended by SCM standards. The main goal of Odyssey-VCS is to aid architects in the concurrent modeling of software systems using heterogeneous UML-based upper CASE tools.

Odyssey-VCS maintains a per-project behavior descriptor that informs how each UML model element type should be dealt. This behavior descriptor determines when evolution information is needed for a UML model element, considering this element as a CI. This evolution information comprises a unique version identification and auxiliary contextual information, such as who changed the element, when it was changed, and why it has been changed. Moreover, this behavior descriptor also indicates which elements are considered atomic for conflict detection purpose. Odyssey-VCS raises a conflict flag when two or more developers concurrently change an element that is considered atomic.

During the design of our approach, we provided our own solutions to overcome some challenges described in the SCM literature [5], such as: (1) data model that deals with complex CIs; (2) homogeneous versioning for different types of CIs; (3) distributed and heterogeneous workspaces; and (4) concurrent engineering with high level models. Moreover, a guiding philosophy of our work is to adopt standardized solutions and successful technologies used in other VCSs.

The rest of this paper is organized as follows. Section 2 details the problem to ground the ensuing discussion. Section 3 presents an overview of Odyssey-VCS, which is followed by a discussion of its internal mechanisms in Section 4. Section 5 shows a straightforward example that demonstrates how the challenges formerly discussed are addressed. Section 6 discusses related work, and we conclude the paper in Section 7 with an outlook at our future work.

2 Problem Statement

VCSs that use a data model based on file system structures usually consider files and directories as their CIs. This approach leads to three types of CIs: composite, textual and binary. The first type, composite CI, is represented by directories and can aggregate textual, binary or other composite CIs. The second type of CI, represented by text files, is the most important type because it can be internally manipulated by the VCS in order to execute basic version control operations, such as diff, patch and merge.

The third type, binary files, is only controlled, but not internally manipulated by the VCS since their internal structures are usually opaque to this tool.

For this reason, text files are seen by these VCSs as white box artifacts, and binary files are seen as black box artifacts. In fact, the use of automatic merge facilities, even for text files, is known to be an error prone activity [12]. They usually provide generic merging algorithms that do not take into account the specific syntactic structure of the text file. In spite of this limitation, almost all VCSs use text file mergers without differentiating text file contents. For example, the same merge algorithm used with a flat text file is also used with a Prolog file, a Java file, a LaTeX file or even an XML file.

Moreover, a common algorithm used by many tools to discover the type of a file is based on control characters. This algorithm searches for a zero ASCII byte inside the file. If this byte is found, the file is marked as a binary; otherwise, the file is marked as a text. This algorithm is used, for example, by MS Visual SourceSafe [23]. Beyond other problems related to the non existence of zero ASCII byte control code in some binary files, all different kinds of text files will be marked as a flat text file. Nevertheless, few VCSs such as Rational ClearCase provide special support for different file types (eg. mdl, doc, xml, etc.) and allow the usage of external mergers [26].

When a file is marked as a flat text file, the VCS considers a line as the unit of comparison[1] (UC). As shown in Fig. 1.a, the UC used in a flat text file is mapped into a paragraph. This is a well fitted mapping because a paragraph has enough cohesion and relative low coupling with other paragraphs, and a defined structure composed by a topic sentence and some supporting sentences. However, this is not the case in other situations as follows:

- A Prolog file usually has complex facts and rules written using more than one line. In this case, UC should be Prolog predicates.
- A Java file, as any other object-oriented language, has complex structures such as packages, classes, methods, and attributes, with methods and attributes as possible candidates to be UC.
- A LaTeX file does not use carriage return and linefeed as delimiters to a paragraph structure. A blank line is needed to identify a paragraph. As a consequence, UC should be the whole structure between blank lines.
- Finally, an XML file is a document composed of elements that may have other elements and attributes by themselves. In this context, a reasonable UC would be elements or attributes. An element or an attribute may be composed of many lines and changes in any of these different lines should be considered as changes in the same UC.

The use of line as UC is especially applicable in situations when a single line has high cohesion and low coupling with other lines. Lines should not be used as UC in files that employ data models with different abstraction structures. In the case of Java files, if a line metaphor is used, UC is mapped to a non existing Java structure (Fig.

[1] We define unit of comparison as an atomic element used for conflict computation. Conflicts occur when two or more developers concurrently work on any part of the same unit of comparison.

1.b). A Java method may be implemented by more than one line; whereas a line may comprise more than one Java command ended by a semicolon. In addition, a command is usually too excessively coupled to other commands to be considered a UC. Hence, the indicated structure should be java attributes and methods.

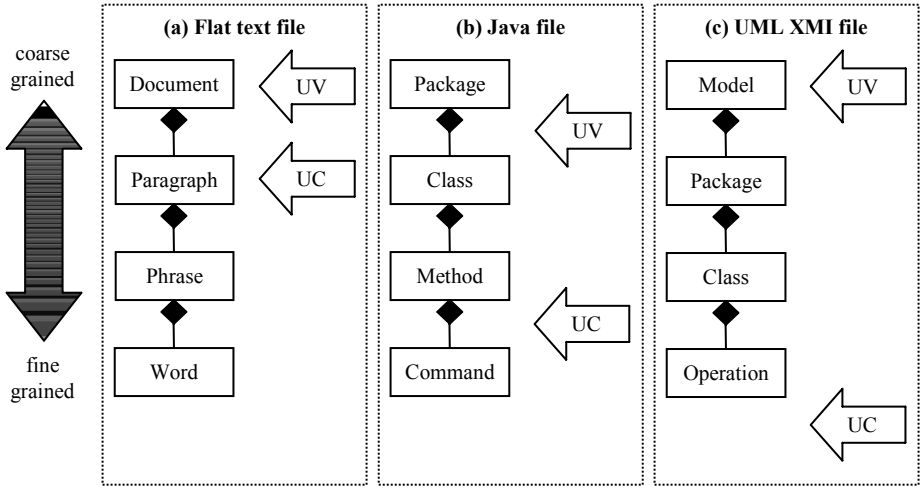


Fig. 1. Current UC applied to flat text files, Java files and UML XMI files.

As discussed before, some VCSs, such as Rational ClearCase, provide support for different abstraction structures through a pluggable merge facility. This strategy, however, is not enough to provide this kind of support because the UC concept only helps to set a boundary between file parts. In these systems, the whole file is considered to be a unit of versioning[2] (UV). For example, in a flat text file, UC is a paragraph and the UV is the whole document, as shown in Fig. 1.a.

This problem is more significant when the file type does not fit the abstraction structure of the file system data model. For example, Java may have one or more classes per file and these classes must be part of the same package, consequently, the UV map to a non existing java element, as shown in Fig. 1.b. In this scenario, a package may have more than one UV distributed through different files, and a class may share the same UV with other classes in the same file.

Java files are not the worst case. A convention may be established to recommend the construction of only one class per file. In this way, UV would be mapped to the class abstraction. Therefore, each class in the system would have their evolution controlled individually. On the other hand, object oriented data model can also be made persistent through files. One of the most common approaches to map an object-oriented data model to a file is to use markup languages. In this case, the whole object network is mapped into a singular file: an XML file. For instance, when Rational ClearCase marks a file as XML, UC is changed and the merge and diff tools act in a

[2] We define unit of versioning as an atomic element associated to versioning information. A new version of the element is created when any part of it is modified.

special way to provide more control over parallel development. However, UV remains being the whole XML file.

In our specific case, UML-based upper CASE tools use an object-oriented data model named Meta Object Facility (MOF) [18] and persist their models using XML Metadata Interchange (XMI) format [20]. In this scenario, the whole object network, composed of thousands of analysis and design artifacts, is persisted into a single XMI file. Fig. 1.c shows a fragment of an UML class diagram mapped to an XMI file. UC is smaller than an operation even if Rational ClearCase XML merge is used. This occurs because a UML operation is described via many XML elements, which represent visibility, return type, arguments, etc., and the parallel development over different XML sub-elements of the same UML operation would not conflict.

Another bad aspect related to UC in this scenario is regarding the proximity principle. UML uses an N-dimension structure composed by different diagrams to model software, and this N-dimension structure is mapped to a single XMI file, which is a one-dimension structure. This problem leverages the difficulty of implementing generic conflict detection algorithms. For example, two classes connected via inheritance association are considered "near" in a UML model. However, if more than one developer changes these classes concurrently, VCS would probably not be able to detect a conflict because the classes are apart from each other in the XMI file.

The problems are even worse when analyzing the UV. UV in this case is the whole UML model and it is not possible to distinguish versions of its parts, such as classes or use cases. The upper CASE tools will only be able to select versions of the whole model to work on. Therefore, the developers could not ask the VCS who changed a specific use case two days ago or what the existing versions of a given class are.

3 Odyssey-VCS Overview

Aiming to diminish the effects of the problems presented in Section 2, we introduce Odyssey-VCS, a flexible VCS for UML model elements. In this section we discuss the high-level features of Odyssey-VCS and show how these features help to overcome the challenges presented in Section 1.

3.1 Complex Data Model

Every upper CASE tool splits modeling elements into two categories: semantic and syntactic elements. Semantic elements symbolize conceptual elements and contain all information related to these elements, while syntactic elements are representations of semantic elements inside a diagram and their data are diagram dependent, like color, position, and size. In the context of the proposed approach, CIs are semantic elements of UML-based upper CASE tools. To be more precise, any subtype of *ModelElement* in the UML meta-model is a candidate to be CI in our approach. For this reason, Odyssey-VCS is able to version even the relationships among UML model elements, since relationships are also model elements. Examples of these model elements are: use cases, actors, classes, class associations, operations, attributes, components, etc.

Due to the complexity of this data model, it is not desirable to have a single versioning behavior for every CI type. Moreover, most SCM standards recommend the definition of a per-project SCM plan [10] and an important section of the SCM plan is the CI identification. The CI identification section of the SCM plan describes all CIs that should be placed under SCM. However, the current VCSs do not work with fine-grained CIs. As a consequence of this problem, all artifacts are put under version control, resulting in an extra overhead to the overall process, since some artifacts are not supposed to be controlled.

Our approach allows a fine-grained definition of CIs. For example, a class may be defined as an atomic CI for a given project; whereas operations and attributes may be controlled in another project. This flexibility provided by Odyssey-VCS allows more precise definition of CIs, adhering to the recommendations of existing SCM standards.

3.2 Homogeneous Versioning

As discussed before, the software development process deals with different kinds of artifacts, such as: use case descriptions, use case diagrams, class diagrams, sequence diagrams, code, test plans, test data, etc. All these artifacts must be controlled in a consistent way to provide snapshots of the system in different moments of development and maintenance. These snapshots when applied to a formal revision are called baselines or system configurations.

A baseline can be seen as a version of the whole system. Each model element has its own version, but the aggregation of these model elements has another version: the baseline version. A problem related to most VCSs is the way they manage baselines. Baselines are not seen as composite versions, but a structure with a completely different semantic. It is a trouble in situations where only one baseline level is not sufficient. Therefore, most current VCSs do not use version and baseline in a homogeneous way due to a lack of composite CIs.[3] A baseline is a special kind of CI, a composite one. It aggregates other CIs and its version is related to the versions of its parts. If a new version is created for some part of a composite CI, a new version should also be created for the whole CI.

In our approach both baselines and versions are dealt in the same way. If a CI is not composed of other CIs, the notion of version is the conventional one. However, if there is a composition relationship between CIs, the version of one CI depends on the version of the other. This situation is comparable to the usage of baselines. The main difference between the conventional approach and our approach is that in our case the baseline is not another type of element, but a CI, too. For example, a UML model has packages composed of classes and classes composed of attributes and operations. In this scenario, a package can be seen as a baseline of all its classes, and a class can be seen as a baseline of its attributes and operations. If one attribute is changed, a new version of the class that encapsulates this attribute is also created, because the class has been indirectly changed. Due to the new version of the class, the package that contains this class will also receive a new version.

[3] As an exception we can cite Subversion [3], which treats directories as composite CIs.

This homogeneous way of treating baselines and versions allows future queries over a specific package or class and complete reconstruction of any previous state, with the correct set of attributes and operations. For instance, it is possible to ask for the most recent version of the root CI of a system (i.e. UML model). Due to the uniform metaphor for versions and baseline, it is easy to transform this element into a part of a bigger system because the whole system is seen by Odyssey-VCS as an ordinary CI.

A possible drawback of this approach is the risk of an explosion of versions of composite CIs. However, file-based VCSs that use this approach deal with this problem by applying hard-links to sub-CIs that have not changed. We also use this technique to avoid waste of storage space.

3.3 Distributed and Heterogeneous Workspaces

The usage of a universal format is a key feature to support heterogeneous workspaces, maintained by different upper CASE tools. XMI is the most adopted format for both commercial and academic UML-based upper CASE tools. For this reason, Odyssey-VCS approach uses XMI as the protocol of communication between upper CASE tools and the VCS. These tools can connect to Odyssey-VCS through the Internet and query for a specific version of a CI, modify it and send back to Odyssey-VCS.

3.4 Concurrent Engineering

Odyssey-VCS is based on an optimistic strategy for concurrency control. The optimistic strategy lets developers change the same model in parallel, and merge the changes when the models are checked-in, as shown in Fig. 2. This strategy leverages parallel work, but increases the complexity of merge algorithms, as detailed later in Section 4.2.

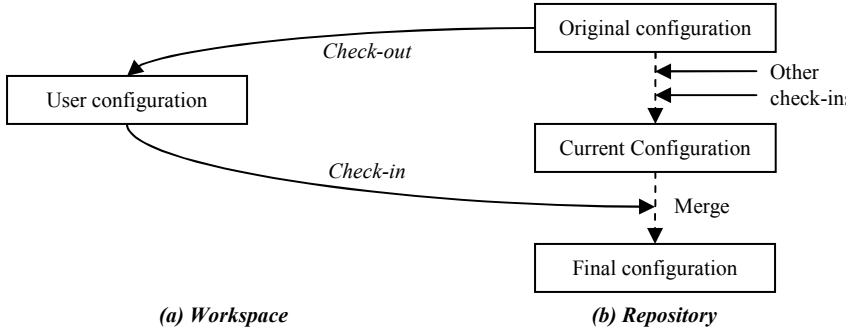


Fig. 2. Optimistic strategy for concurrency control

The original configuration shown in Fig. 2.b is the starting point of a new development cycle. The User configuration is created when a given developer checks-out the

original configuration and performs some changes. During this period of time, other developers also work on the original configuration, merging their work into the current configuration. Finally, the user configuration is also merged into the current configuration, creating the final configuration.

When a conflict is detected during the merge procedure, the whole check-in is rolled-back and the developer receives a message containing a detailed conflict description and the original, user and current configurations. After performing a manual merge, which can be supported by external tools, the developer resubmits the UML model to the repository.

4 Odyssey-VCS internals

The Odyssey-VCS architecture, which was implemented in Java from the scratch to avoid dependencies to existing file-based VCSs, is composed of three major layers: client, transport, and server. The most important element of the *client layer*, presented in Fig. 3.a, is the upper CASE tool. We are assuming that this tool uses UML as modeling notation and is able to externalize UML models using XMI. The integration between the upper CASE tool and the Odyssey-VCS infrastructure can be done via two alternative mechanisms: Odyssey-VCS plug-in and Odyssey-VCS client tool. Some upper CASE tools offer an extension infrastructure that allows the addition of external tools. In this case, Odyssey-VCS plug-in can be used, providing a seamless integration. For instance, we adopted this mechanism to integrate Odyssey-VCS with Odyssey environment [25]. However, some upper CASE tools have a poorly documented extension infrastructure, or do not even have it. In these situations, it is possible to use the Odyssey-VCS client tool. This tool opens an XMI file previously saved by the upper CASE tool and allows the execution of Odyssey-VCS commands. This mechanism was used to integrate Odyssey-VCS with Poseidon [1].

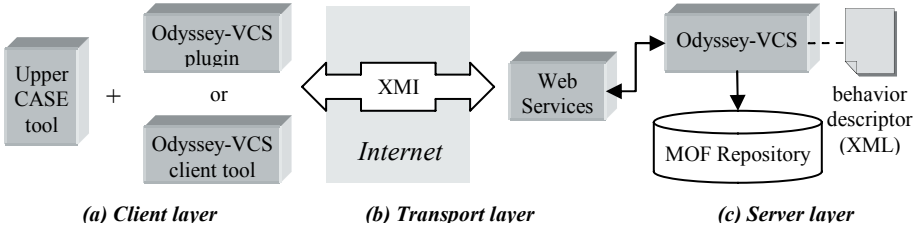


Fig. 3. Odyssey-VCS Overall Architecture

The *transport layer*, presented in Fig. 3.b, is responsible to allow distributed development of UML models over the Internet. The current implementation of this layer uses local calls or Web Services [2] as a transport protocol. However, it can be replaced by other protocols, such as WebDAV, RMI, sockets, etc. Moreover, before being sent over the Internet, the XMI files pass through a compression layer (zlib) to increase the overall throughput of the transport layer. This kind of approach, in place of using deltas, is also being adopted by other SCM tools [6].

Finally, the *server layer* processes the XMI files, applying different versioning behaviors depending on the project needs. The checked-in XMI file is transformed into an object network, which is merged with existing objects and stored into a MOF Repository named MDR [13] for further querying and retrieval. Both behavior configuration and merge algorithm are presented in the next sections.

4.1 Behavior Configuration

When a UML model element is checked-in, a specific action should be performed. However, as a flexible approach, Odyssey-VCS reads a behavior descriptor to decide what to do. This behavior descriptor informs which elements are UC and UV. For example, the scrap of a behavior descriptor shown in Fig. 4 is indicating that no versioning information should be stored for attributes and operations. On the other hand, classes are considered as CIs (UV=true), meaning that every version should be registered. Moreover, classes are also considered atomic elements (UC=true). Due to that, Odyssey-VCS will notify a conflict when two or more people edit the same class, even if they are working in different parts of the class.

```
<type name="org.omg.uml.foundation.core.UmlClass">
        <UC>true</UC>
        <UV>true</UV>
</type>
<type name="org.omg.uml.foundation.core.Attribute">
        <UC>true</UC>
        <UV>false</UV>
</type>
<type name="org.omg.uml.foundation.core.Operation">
        <UC>true</UC>
        <UV>false</UV>
</type>
```

Fig. 4. Scrap of a behavior descriptor XML file

It is important to notice that every software development project has its own behavior descriptor. This allows the customization of Odyssey-VCS to the specific needs of projects. For instance, if a project does not define class as UC, no conflict is detected when two people work on different parts of it. On the other hand, if operation is set as UV, every change on operations is registered together with versioning information.

Another important aspect is the interplay between UV and non-UV elements. Odyssey-VCS stores all physical versions of every element, but only stores logical versioning information of UV elements. For this reason, it is possible to correctly retrieve the context relationships of a UV element, even if it is related to non-UV elements.

4.2 Merge Algorithm

A built-in merge algorithm is also provided together with the flexible versioning infrastructure. This merge algorithm takes into account the configurations shown in Fig.

2. After analyzing the presence/absence of elements in these configurations, and the internal values of these elements after a check-in, we reached a complex scenario that is summarized in Table 1. The configurations and relations among them, used in Table 1, are defined as follows:

- O: Original configuration;
- U: User configuration;
- C: Current configuration;
- F: Final configuration;
- e_X: Element "e" in configuration "X"; and
- $e_X \equiv e_Y$: True if element "e" is identical in both configurations "X" and "Y".

Table 1. Odyssey-VCS merge algorithm

Case	$e \in O$	$e \in C$	$e \in U$	$e_O \equiv e_C$	$e_O \equiv e_U$	Action
1	T	T	T	T	T	Add e_C (or e_U) into F
2	T	T	T	T	F	Add e_U into F
3	T	T	T	F	T	Add e_C into F
4	T	T	T	F	F	Notify a conflict: "concurrent changes over the same element"
5	T	T	F	T	N/A	None (do not add "e" into F)
6	T	T	F	F	N/A	Notify a conflict: "concurrent removal and change over the same element"
7	T	F	T	N/A	T	None (do not add "e" into F)
8	T	F	T	N/A	F	Notify a conflict: "concurrent removal and change over the same element"
9	T	F	F	N/A	N/A	None (do not add "e" into F)
10	F	T	T	N/A	N/A	N/A
11	F	T	F	N/A	N/A	Add e_C into F
12	F	F	T	N/A	N/A	Add e_U into F
13	F	F	F	N/A	N/A	N/A

Table 1 shows, for every possible scenario, which action should be taken by Odyssey-VCS. For example, case 3 shows a scenario where a given element (use case, for example) exists in all configurations ($e \in O$, $e \in C$, and $e \in U$), was changed in the current configuration ($e_O \not\equiv e_C$), but was not touched in the user configuration ($\mathbf{e_O \equiv e_U}$). In this case, Odyssey-VCS promotes the element from the current configuration to the final configuration. On the other hand, case 6 shows a scenario where a given element (an operation, for example) was removed from the user configuration ($e \notin U$) but exists in all other configurations ($e \in O$, $e \in C$). However, the element was changed by other users ($e_O \not\equiv e_C$). As a result to this scenario, Odyssey-VCS notifies a conflict, arguing that the same element was removed and changed by different developers.

The results of the merge algorithm are consistent to the UML structure (guaranteed by MDR repository), but may be inconsistent with UML well-formedness rules. The current release of Odyssey-VCS does not apply well-formedness rules consistency check. However, this validation can be obtained by external tools.

5 Example

In this section we present an intentionally simple usage example that aims to illustrate how our approach provides flexibility and concurrent access during the evolution of UML models. The target system of this usage example is a hotel network control system presented in the literature.

Initially, suppose that Odyssey-VCS is configured as described in Table 2. It is important to notice that *Class*, a composite element that encloses *Attributes* and *Operations*, is configured as UC. In other words, this means that should two or more developers interact concomitantly over the same class, a conflict will happen. This conflict occurs even if they are working on different parts of the class. Moreover, *Actor* is not configured as UV. This means that no versioning information will be stored regarding this element.

Table 2. Odyssey-VCS configuration

Element	Unit of Versioning (UV)	Unit of Comparison (UC)
Model	True	False
Package	True	False
Class	True	**True**
Attribute	True	True
Operation	True	True
Use Case	True	True
Actor	**False**	True

After configuring the Odyssey-VCS system, an initial version of the model has been committed to the repository. This initial version, shown in Fig. 5, comprises six classes, all in the same package, four use cases, and one actor. This model is used as the basis for further development.

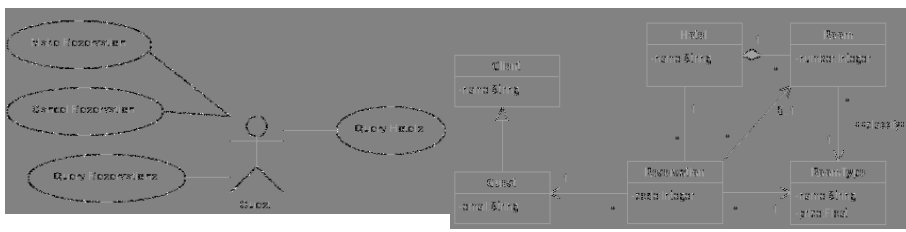


Fig. 5. Hotel network control system use case and class models

In the following, two developers, namely John and Mary, are using different upper CASE tools, respectively Poseidon and Odyssey, to concurrently change the initial version of the hotel network control system model. John wants to rename the *email* attribute of the *Guest* class to *telephone*, add the *gender* attribute into the *Client* class and add a new use case named *Modify Reservation*. On the other hand, Mary wants to change the type of the *price* attribute of the *RoomType* class from *Float* to *Currency* and include two new operations in the *Guest* class: *getEmail():String* and *setEmail(email:String):void*.

The changes performed by John, shown in Fig. 6.a, were committed first into the repository. No conflicts were detected and the commit was successfully merged into the current version of the repository. After John's commit, the current version of *Guest* class in the repository has no more the *email* attribute. However, the version of the *Guest* class in the Mary workspace is out-of-date, still containing the *email* attribute, as shown in Fig. 6.b.

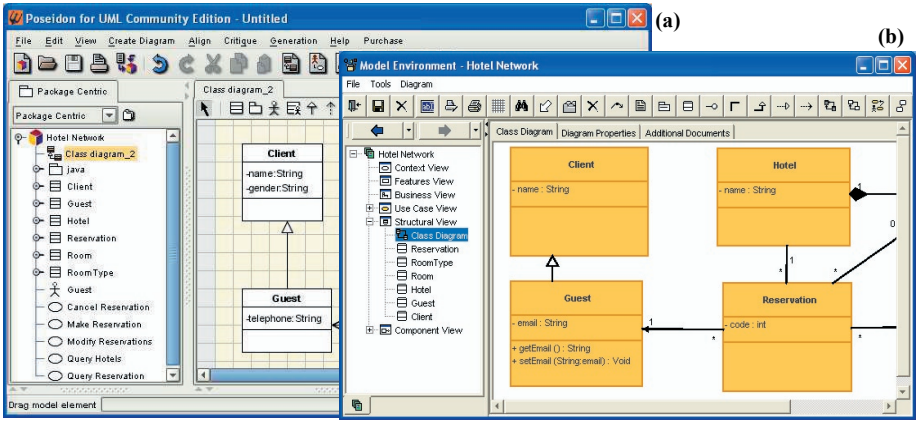


Fig. 6. Poseidon (a) and Odyssey (b) working over the same UML model.

When Mary tries to commit her version, all UML model elements, but *Guest* class, are correctly merged. Even the sub-elements of the *Guest* class were individually merged successfully. However, besides their syntactical correctness, they are semantically incompatible, because the operations *getEmail():String* and *setEmail(email:String):void* were created to manipulate the *email* attribute, which was renamed to *telephone* by John. Fortunately, *Class* type was defined as UC. Due to that, a conflict is raised, as shown in Fig. 7.

Odyssey-VCS provides all necessary information to allow Mary fixing the conflict. This information comprises, in addition to Mary's local version of the model, the original and the current versions of the repository. After manually fixing the conflict, Mary is finally able to commit her changes into the repository. Fig. 8.b shows the final state of the repository, which has the original version of the model (version 1), John's commit (version 2) and Mary's commit (version 3).

Fig. 8.a shows the third version of the model in Mary's workspace, after a new check-out. This third version contemplates the original intention of both John and Mary. It is worth to notice that every version that is relative to a type defined as UV in Table 2 has contextual information when stored in the repository. This contextual information, which encloses date, responsible, and additional comments, can be further used by other developers. However, *Actor* was not defined as UV. For this reason, its versions are neither shown in Fig. 8 nor computed by Odyssey-VCS.

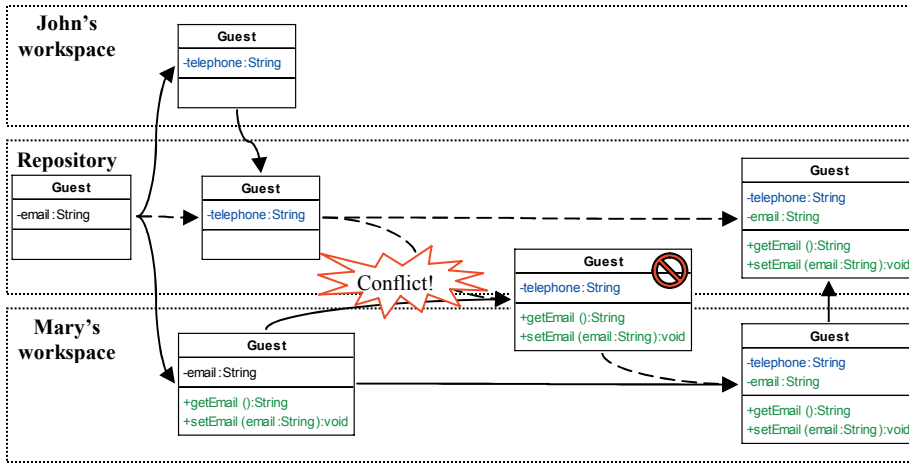


Fig. 7. Merge and conflict detection scenario

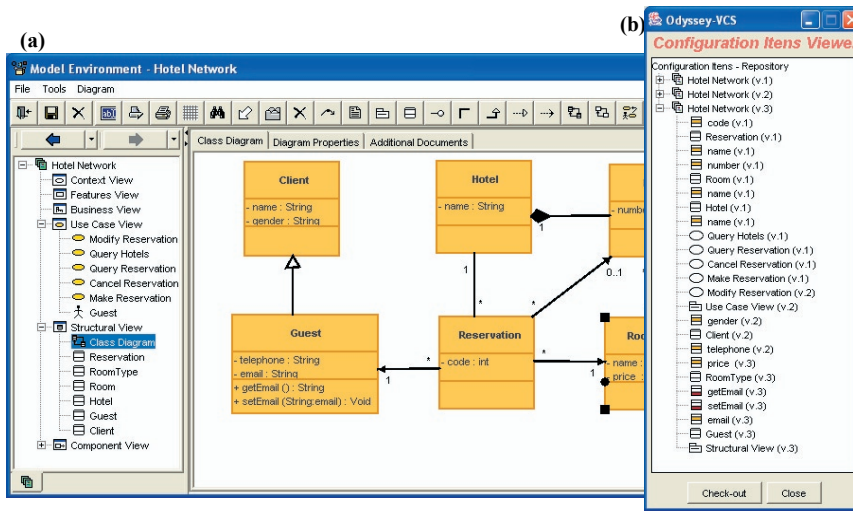


Fig. 8. Odyssey (a) performing check-out through Odyssey-VCS plugin (b)

6 Related Work

Most commercial and open-source VCS are based on file system data model [3, 7, 23, 26]. As previously discussed, these VCS have several limitations to manipulate artifacts with complex internal data model. However, these solutions are generic and mature, being suitable for versioning of text-based artifacts. We do not see these approaches as direct competitors of our approach. Our approach can be used to version

UML model elements while these approaches can be used to version source-code in the same software development project.

There are also other approaches that employ other data models, such as entity-relationship or even object oriented. However, these approaches work at source code level and are focused on a specific programming language. For instance, Goldstein et al. [8], Habermann et al. [9], and Render et al. [22] support versioning of Smalltalk, C and Pascal source code, respectively. Our approach can also be seen as complementary to these approaches since we are focused on UML model elements and these approaches are focused on source-code.

Some few approaches use non file system data model to version analysis and design artifacts. For instance, Ohst et al. [17] propose an approach for versioning analysis and design artifacts via syntax trees stored in XML files. Working at fine grained UML artifacts they can correctly manage structural changes on these artifacts. However, the usage of XML does not mean adherence to modeling standards. Their XML format does not follow XMI specification for UML, leading to incompatibilities with existing UML-based upper CASE tools. Moreover, Nguyen et al. [16] use a hyper-versioning system to apply version control over complex artifacts, including UML analysis and design artifacts. This work has a strong focus on versioning relationships among the elements. However, it is also based on a proprietary UML data model, reducing compatibility with existing upper CASE tools.

Finally, OMG is working on a specification for MOF versioning [19]. Besides the nonexistence of a final specification version up to now, it is possible to notice that Odyssey-VCS is pretty adherent to the specification philosophy. Similar to the specification, Odyssey-VCS has its own versioning meta-model. Moreover, it stores the versioned elements into separate per-version extents with associated history of changes. Probably, it will be straightforward to adhere to the final version of the specification in the future.

7 Conclusion

In this paper we presented an approach for version control of UML model elements. Our approach differs from the existing approaches in the following aspects. First, we provide support for flexibility during CI identification, allowing the configuration of UC and UV for UML model elements. Second, our approach is based on well adopted standards, raising the compatibility with existing upper CASE tools. Finally we have focused on current challenges of SCM to avoid reinventing the wheel regarding already solved problems. Besides these main aspects, our approach also provides a built-in merge algorithm, supporting concurrent development.

The existence of a fine-grained VCS for UML model elements can be seen as the basis for upcoming work. For instance, Dantas et al. [4] have proposed an approach for traceability link detection via data mining over UML model elements stored in the Odyssey-VCS repository. Moreover, another work is being performed to transform Odyssey-VCS into a change-oriented VCS [15]. All these tools are part of a broader

infrastructure named Odyssey-SCM [14], which aims to provide SCM functionalities for component-based development environments.

Our approach, however, is currently tightly coupled to the UML meta-model. An important future work is the generalization to the MOF meta-model layer, allowing versioning of any MOF compliant meta-model. Additionally, the current version of Odyssey-VCS does not use deltas to compose versions. On one hand, the negative impact of this decision is higher network traffic. On the other hand, we do not need to compute a version based on prior versions and deltas, which saves some CPU cycles. We also use zlib to help reducing the transport overhead due to the absence of deltas. While performance and scalability were not a major concern for this first prototype, we intend to work on these issues for the next releases. Currently, we are performing some benchmarks to measure the performance of Odyssey-VCS when the size of the repository increases. After that, we intend to run some case studies in real software development scenarios.

Moreover, another limitation is the use of a predefined merge algorithm. Future releases of Odyssey-VCS should allow the replacement of the built-in merge algorithm for project-specific merge algorithms. Another future work is the construction of a tool that allows visual merge of UML models. The current version of Odyssey-VCS only notifies the conflict, providing all information for the merge. However, the merge itself is not supported by Odyssey-VCS, requiring the user to do it directly in the XMI file or using existing UML-based upper CASE tools.

Acknowledgments

Our thanks to the members of the Software Reuse Group at COPPE/UFRJ, especially Cristine Dantas and Luiz Gustavo Lopes, who helped in many discussions regarding the architecture of the approach. Moreover, we would like to thank CNPq and CAPES for the financial support.

References

1. Boger, M., Sturm, T., Schildhauer, E., and Graham, E.: Poseidon for UML user guide. Gentleware AG (2000)
2. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D.: Web Services Architecture - W3C Working Group Note. World Wide Web Consortium (W3C). In: http://www.w3.org/TR/ws-arch, Accessed in: 25/Jul/2005
3. Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M.: Version Control with Subversion. O'Reilly (2004)
4. Dantas, C. R., Murta, L. G. P., and Werner, C. M. L.: Consistent Evolution of UML Models by Automatic Detection of Change Traces. International Workshop on Principles of Software Evolution (IWPSE), Lisbon, Portugal, September (2005)
5. Estublier, J.: Software Configuration Management: a Roadmap. International Conference on Software Engineering, The Future of Software Engineering, Limerick, Ireland, June (2000) 279-289

6. Estublier, J., Leblang, D., Clemm, G., Conradi, R., Tichy, W., van der Hoek, A., and Wiborg-Weber, D.: Impact of the research community on the field of software configuration management: summary of an impact project report. ACM SIGSOFT Software Engineering Notes, Vol. 27, no. 5, September (2002) 31-39
7. Fogel, K. and Bar, M.: Open Source Development with CVS. The Coriolis Group, Scottsdale, Arizona (2001)
8. Goldstein, I. P. and Bobrow, D. G.: A Layered Approach to Software Design. In: Barstow, D. R., Shrobe, H. E., and Sandewall, E. (eds.): Interactive Programming Environments. McGraw-Hill, New York, NY (1984) 387-413
9. Habermann, A. N. and Notkin, D.: Gandalf: Software Development Environments. Transactions on Software Engineering, Vol. 12, no. 12, December (1986) 1117-1127
10. IEEE: Std 1042 - IEEE Guide to Software Configuration Management. Institute of Electrical and Electronics Engineers (1987)
11. ISO: ISO 10007, Quality Management - Guidelines for Configuration Management. International Organization for Standardization (1995)
12. Leon, A.: A Guide to Software Configuration Management. Artech House Publishers, Norwood, MA (2000)
13. Matula, M.: NetBeans Metadata Repository. NetBeans Community. In: http://mdr.netbeans.org, Accessed in: 25/Jul/2005
14. Murta, L. G. P., Dantas, C. R., Oliveira, H. L. R., Lopes, L. G. B., and Werner, C. M. L.: Odyssey-SCM. In: http://reuse.cos.ufrj.br/odyssey/scm, Accessed in: 25/Jul/2005
15. Murta, L. G. P., Oliveira, H. L. R., Dantas, C. R., Lopes, L. G. B., and Werner, C. M. L.: Towards Component-based Software Maintenance via Software Configuration Management Techniques. Workshop on Modern Software Maintenance (WMSWM), Brasília, Brazil, October (2004)
16. Nguyen, T. N., Munson, E. V., and Boyland, J. T.: The molhado hypertext versioning system. Conference on Hypertext and Hypermedia, Santa Cruz, USA, August (2004) 185-194
17. Ohst, D. and Kelter, U.: A Fine-grained Version and Configuration Model in Analysis and Design. International Conference on Software Maintenance (ICSM), Montreal, Canada, October (2002) 521-527
18. OMG: Meta Object Facility (MOF) Specification, version 1.4. Object Management Group. In: http://www.omg.org/technology/documents/formal/mof.htm, Accessed in: 25/Jul/2005
19. OMG: MOF 2.0 Versioning and Development Lifecycle RFP. In: http://www.omg.org/cgi-bin/doc?ad/02-06-23, Accessed in: 25/Jul/2005
20. OMG: XML Metadata Interchange (XMI) Specification, Version 2.0. Object Management Group. In: http://www.omg.org/technology/documents/formal/xmi.htm, Accessed in: 25/Jul/2005
21. Pressman, R. S.: Software Engineering: A Practitioner's Approach. McGraw-Hill (1997)
22. Render, H. and Campbell, R.: An Object-oriented Model of Software Configuration Management. International Workshop on Software Configuration Management, Trondheim, Norway, June (1991) 127-139
23. Roche, T. and Whipple, L. C.: Essential SourceSafe. Hentzenwerke Publishing (2001)
24. Voelcker, J.: Automating Software: Proceed with Caution. IEEE Spectrum, Vol. 25, no. 7, July (1988) 25-27
25. Werner, C. M. L., Mangan, M. A. S., Murta, L. G. P., Souza, R. P., Mattoso, M., Braga, R. M. M., and Borges, M. R. S.: OdysseyShare: an Environment for Collaborative Component-Based Development. IEEE Conference on Information Reuse and Integration (IRI), Las Vegas, USA, October (2003) 61-68
26. White, B. A.: Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction. Addison-Wesley (2000)

Model Data Management – Towards a common solution for PDM/SCM systems

Jad El-khoury

Royal Institute of Technology (KTH), Mechatronics Division, Machine Design, Sweden
jad@md.kth.se

Abstract. Software Configuration Management and Product Data Management systems have been developed independently, but recently the need to integrate them to support multidisciplinary development environments has been recognised. Due to the difference in maturity levels of these disciplines, integration efforts have had limited success in the past. This paper examines how the move towards model-based development in software engineering is bringing the discipline closer to hardware development, permitting a tighter integration of their data management systems. An architecture for a Model Data Management system that supports model-based development is presented. The system aims to generically handle the models produced by the different tools during the development of software-intensive, yet multidisciplinary, products. The proposed architecture builds on existing technologies from the mature discipline of mechanical engineering, while borrowing new ideas from the software domain.

1 Introduction

Organisations involved in the development of large and complex products need to deal with a large amount of information, created and modified during the development and product life cycle. To support this need, an organisation normally adopts some kind of product management environment. Many such management solutions are currently available, and it is generally the case that each tends to focus on a specific class of products, determined by the major engineering domain involved in the product development. The development of software-intensive products relies on Software Configuration Management (SCM) systems, while mechanical system development uses Product Data Management (PDM) systems.

In the development of products that involve the collaboration of various engineering disciplines, a number of these management environments come into simultaneous use. This is necessary since developers from each discipline require the specific support provided by its corresponding system. An automotive system is a typical such product, where traditional engineering disciplines such as control, software, mechanical and electrical engineering, need to interact to meet the demands for dependable and cost-efficient integrated systems.

Considering the central role these environments take in controlling the development process as well as facilitating the communication between developers, integrating them becomes essential for the successful integration of the efforts of all disci-

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

plines involved. In multidisciplinary development, allowing the environments to run unsynchronised creates a source of inconsistencies and conflicts between the disciplines. In other words, it is equally important to provide (where possible) a common set of support mechanisms and principles within, as well as between, the disciplines.

While most of the general facilities provided by these solutions overlap, variations in the details exist due to the differing needs of the domains. This leads to complications and difficulties when attempting to integrate them [1]. In this paper, we discuss how the move towards a model-based development approach in software engineering is bringing it closer to the hardware engineering discipline, allowing for a tighter integration of their management systems. We advocate a common model-based management system that borrows from the technologies of each of these disciplines. In the next section, we discuss the differences between conventional SCM and PDM tools and investigate the effect of adopting model-based development in software engineering in bringing these solutions closer. Section 3 presents a management system architecture that takes advantage of this change. This is followed by a discussion of related work in the area of PDM/SCM integration, before concluding the paper in section 5.

2 Model-based Development – Bringing Software development towards Hardware Development

Model-based development (MBD) refers to a development approach whose activities emphasise the use of models, tools and analysis techniques for the documentation, communication and analysis of decisions taken at each stage of the development lifecycle. Models can take many forms such as, (but not limited to,) graphical, textual and prototype models. It is essential however that the models contain sufficient and consistent information about the system, allowing reproducible and reliable analysis of specific properties to be performed.

With the maturity of the software discipline, the need to move towards a more model-based development approach is being recognized. This need is exemplified in (but certainly not limited to) the OMG efforts [2][3], and the wide range of tools supporting them.

In this section, we will investigate how the adoption of model-based development in software engineering can help bridge a gap between software and hardware development, leading towards a common solution for the data management of multidisciplinary products. In [1], three crucial factors for a successful integration of PDM and SCM are presented: processes, tools and technologies and people. We follow this categorisation in this investigation. New challenges facing such a common solution are also discussed.

2.1 Processes

The difference in the development process of software and hardware products has been most influential in the divergence between their management tools. The more mature hardware development expects support during the complete product life cycle

from the early concept design phases down to manufacturing and post-production phases [4]. All product data from all these phases is expected to be handled and related through the PDM system. In comparison, as with any new discipline, early software development occurred in a relatively more ad-hoc manner with no, or little, early design and analysis phases. Consequently, these early phases were beyond the scope of SCM tools [4][5], and SCM was only expected to manage the large amount of source files produced during the implementation phase of software development.

In software engineering, the application of the model-based approach throughout the complete development process implies the need to handle different kinds of documentation from the early design and analysis stages, as well as implementation. Conventional SCM tools have so far incorporated these additional documents by simply treating them as files, without differentiating them from source code files. However, one cannot claim that SCM handles the development process appropriately, since no distinction is made between the types of documents produced during the different development phases. For this to be possible, we argue that the development process itself needs to be reflected in the product information model.

In [1], it is argued that the life cycle processes of the software and hardware development should be integrated for the successful integration of PDM/SCM. The challenges for such integration and a simple solution are then suggested. What seems to be missing in the discussion is how process integration would be beneficial for the integration of PDM and SCM systems. Studying the functionalities of PDM/SCM, one can see that such systems simply provide the infrastructure to enforce a given process (see section 2.2) and play no direct role in integrating the development processes. Instead, PDM/SCM functionalities focus on the product data produced. For this reason, while process integration may be desired within an organisation, for the purpose of integrating PDM/SCM systems, it is even more important to focus on the integration of the outcomes/artefacts produced at each phase of the product lifecycle. The ultimate goal is the tight integration of the hardware and software components of the final product, and not the process of getting there.

2.2 Tools and Technologies

This category is further divided into six basic functionalities expected of PDM/SCM systems: data representation, version management, management of distributed data, product structure management, process support and document management.

Data Representation A major difference between PDM and SCM lies in the kind of data that the support tools are expected to handle [6]. In hardware development, the need to provide a seamless workflow from design to manufacturing phases has forced PDM systems to not only handle the documents produced, but much of their internal contents (metadata) as well. A detailed information model of the product data is an integral part of a PDM system [7]. Software development, on the other hand, has so far adopted a file-based approach, only managing the files produced during development, and where the only relations handled between the files is that of the file system itself (a small amount of meta-data is also handled such as file author and modification date). The internal structure of these files and the semantical

relationships between them has so far been outside the scope of SCM tools. PDM can be interpreted as managing product representations, while SCM manages the final product itself [8].

With the maturity of the software discipline, and its move towards a more model-based development approach, many documents (analysis models, uses cases, etc.) will be produced during development. These documents act as models representing certain aspects of the product and will not necessarily end up in the final product. Nevertheless, the different types of documents need to be identified in the management system and related to specific development stages.

The information stored in the documents is interrelated. For this reason, SCM systems supporting model-based development would need to, not only manage the files storing the models, but also the internal content of these models, allowing fine-grained relationships between the document contents to be setup. An information model of the complete information space contained in the models need to be an integral part of a SCM system.

In a model-based development approach, developers need to be shielded from the file structures used to store the models built, allowing them to focus on the models and their structures. This strategy is adopted by many modern modelling tools that may use database systems to store and hide models, and a modern management system should follow in this track.

Version Management In PDM systems, revisions of an object are manually managed by the user and form a sequential series, with no possibility of performing parallel changes. In contrast, versions in SCM systems form a graph structure, with the possibility to perform branching in the development, followed by merging of the branched tracks. Due to these differences, the later approach facilitates concurrent engineering, which is limited in the former.

Accepting that SCM systems need to focus on modelling items, and not only the files storing these items, it becomes essential for version management functionality in SCM systems to similarly focus on the contents provided in these files. Instead of differentiating between the lines of text in different versions of a file, it is differences between the modelling items in different versions of a model that need to be identified and managed.

Since conventional SCM systems do not handle the internal semantics of files, it has also been out of its scope to ensure that parallel changes to the same item (file) are consistent upon a merge. SCM simply provides the mechanism to branch and merge changes made to unrelated lines of text. The burden is placed on the user to ensure that merged changes from different development tracks are consistent semantically. It was hence relatively easy to provide such semantic-free functionality.

Model-based version management becomes a challenge for SCM systems. Complexity arises due to the different kinds of modelling items that may exist in a model compared to the single type (lines of text) that are conventionally handled. It is no longer possible to provide the exact versioning functionalities for all kinds of documents in the system. In the best case, customisation of a generic mechanism will allow the reuse of much of this functionality.

An additional challenge is to ensure consistent parallel changes to the models stored in the files during version management. While lines of text in a file can be

treated individually, modelling items in a model are generally tightly interrelated. Changes to one item may have implications on other items in the model. This implies that even though each individual set of changes in two parallel change tracks is semantically valid, merging these changes into a consistent set is not as simple as the union of the changes since the relations between the modelling items need to be taken into account. For example, in a class diagram, one track of changes may have deleted a certain class, while in another track a new association is created between that class and another. In merging these changes, it is first necessary to establish if the deleted class needs to be reintroduced before allowing the presence of the new association.

In dealing with this problem, an SCM system can adopt the approach of PDM of disallowing parallel changes and in this way preventing the problem from occurring in the first place. Another approach is to develop branch/merge mechanisms that work on model structures, maintaining support of concurrent development of models for software developers. A successful implementation of the latter approach can also be beneficial for hardware development, where the possibility to concurrently develop models becomes possible, leading the way for new development processes.

The need for concurrent changes to the same source code files partly originates from the less mature adhoc development of earlier software systems before software "engineering" became a discipline. It is argued that a structured model-based development approach would reduce the need for parallel access to the same product data and hence the former approach becomes more appropriate. In the case where concurrent changes remain a necessity, the latter approach needs to be supported.

Nevertheless, branch/merge mechanisms in SCM remain a necessity for the management of product variants. However, in model-based development, this implicit management of variants should be made more explicit, by representing variants in the product information model.

As discussed in section 4, the model-based approach to versioning and branching/merging is gaining ground in the SCM community. In this paper, we advocate taking advantage of this new trend in the integration of PDM and SCM systems.

Management of Distributed Data The need to manage geographically distributed data seems to be common for both disciplines, with the difference being in the technical solution provided by the management systems. PDM systems provided a more limiting functionality by not allowing concurrent access to distributed data. This difference is closely related to that discussed in the previous subsection, and synchronising the earlier difference will naturally lead to the synchronisation of this functionality. Technically, a common solution will choose either the currently adopted PDM or SCM solution based on whether concurrent access is desired or not.

In a model-based approach to distributed data management, the functionality would focus on the management of distributed fine-grained model data items and not the files storing these items.

Product Structure Management In hardware systems, the physical structure of the final product is the single predominant structure. This structure is used throughout the development phases as a basis for the information model to which all other information is related. Conventional SCM systems do not explicitly support the

structure of the product, focusing instead on the directory structure of the files it manages.

In a model-based approach to software development, an SCM system would need to focus on the internal structures of the models stored in the files instead. Unlike hardware products, when using models throughout the development phases, the software structure will vary widely, and hence the product structure management functionality of a model-based SCM needs to handle many different parallel structures. Relationships between these structures will also need to be taken into account.

Given the possibility to manage multiple structures, it becomes easier to also manage products resulting from the integrated effort of hardware and software development. Each discipline would be able to maintain its own structure. The possibility to set up relationships between the structures results in a tight integration of hardware and software components.

Process Support As mentioned in section 2.1, it is necessary to integrate the process support functionalities of PDM and SCM systems. Software and hardware development would need to follow different development processes, and this functionality should be able to support each of the chosen processes, yet based on common fundamental mechanisms: workflow management, user assignment, approach rule mechanisms, etc. As mentioned in [1], such functionality is already quite similar in PDM and SCM systems.

Document Management Document management is an integral part of PDM systems, and such functionality is missing in conventional SCM systems. The need for document management by software developers is apparent, and hence a common efficient support ought to be technically feasible.

2.3 People and Cultural Behaviours

In [9], some of the differences in the terminologies used by software and hardware engineers are highlighted. These differences are attributed to the differences in the development phases generally focused on by these disciplines. For example, in software engineering, “design” is traditionally defined as building a model of the system up to the point at which coding begins. In hardware development, however, “design” would also include broader activities such as requirements and testing activities.

In adopting a model-based approach in both disciplines, and as a by-product of integrating the outcomes of each of the phases of the development processes as advocated earlier, it becomes necessary to integrate the meaning of some of the terminology used.

An important function of models is communication. While models are domain-specific and can only be understood in details by engineers of the specific disciplines, such models can be still used to communicate certain aspects of the design to other engineers, if presented at the right level of abstraction. If models from the various disciplines can be successfully interrelated to form a consistent whole view of the system

through a common management system, such interrelations can also act as interaction points between the disciplines, reducing any risks of inconsistencies and conflicts.

2.4 Conclusion

The fundamental differences between SCM and PDM systems stem from the different needs of the disciplines they aim to support. As software development becomes increasingly model-based, and requires support throughout its development life cycle, its needs become closer to those of hardware development. In particular, the process management and information modelling functionalities expected of SCM systems come closer to those provided by PDM systems for hardware development.

This leads the way for an easier and more effective integrated management platform satisfying the needs of both disciplines using a common set of mechanisms. The management functionality ought to take advantage of the commonality between the disciplines – the use of models – in the development process by focusing on models and their internal content as central entities. This allows the same model-based functionalities to be used by both disciplines. We term such an approach as Model Data Management (MDM).

3 Model Data Management

In this section, we present an architecture for a Model Data Management (MDM) system that aims to generically support and control different kinds of models produced from a set of different tools and disciplines.

3.2 Tool Architecture

The envisaged architecture is shown in figure 1. The platform consists of two main parts: A set of tool-specific adaption layers and a data repository with mechanisms to handle this data. The data repository stores the data for each of the tools. To perform this role in a generic way, the data from the different tools is expected to be presented in a neutral form, and this functionality is provided by the adaption layer. Triggered either by a tool or the repository, the corresponding adaption layer permits the data flow between a tool and the repository, in a predefined format. The following subsections will further discuss these components.

Given its maturity, we aim to base the proposed MDM system on a configurable PDM system. The major advantage of using a PDM system is the possibility to define information models, with a high level query language to access and modify the model data in the repository. These facilities generally do not exist in conventional SCM systems. In addition, it is envisaged that the development of the remaining MDM functionalities is made easier given the already developed functionalities of PDM such as the support for distributed development, change management, workflow control, etc. The adoption of a PDM system is not indispensable and one can envisage building an independent MDM that supports both disciplines.

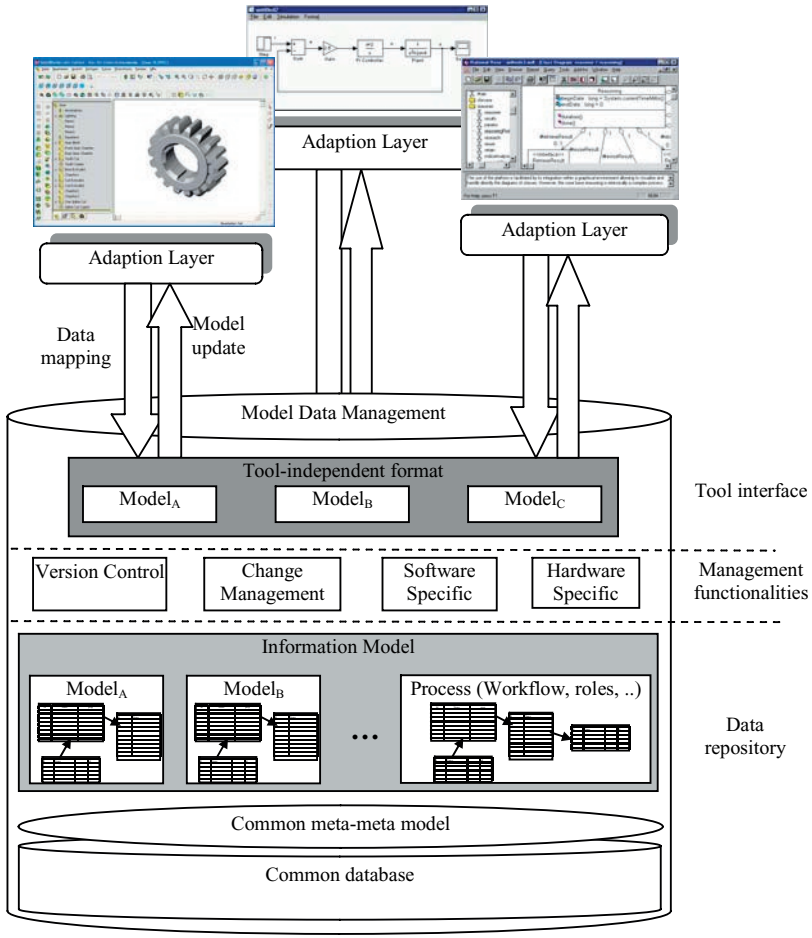


Fig. 1. The major components of the MDM architecture. (Note that the graphical tools are mock-ups shown here for illustration purposes only.)

Data repository The data repository stores the data from each of the tools integrated into the platform. Tool data can be separated into graphical and model data [10] and both types of data need to be managed by the system, giving full control over the models.

It is important to note that the data repository is not expected to be the primary storage medium for each integrated tool, and to which each tool implementation needs to conform. Similar to the a-posteriori approach in [24], an integrated tool is self-sustained, and is only a-posteriori integrated through an adaption layer (See next sub-section).

The content of a model is generally defined using a specific meta-model that reflects its internal structure and constraints of how modelling elements can be com-

bined to form a valid model. In many tools such as in Simulink [11], a meta-model is implicitly assumed, while others, such as any UML tool [3], are strongly based on a given meta-modelling framework.

This meta-model acts as a basis for the data schema used by a tool to internally manage and store the model contents. Similarly, the MDM system managing an integrated model needs to map the corresponding meta-model onto the data schema of the repository. Since different types of models assume a different meta-model, each model type would occupy a separate space in the repository with a different data structure. However, in order to simplify the specification of a schema for each integrated model, a meta-meta-model is adopted as a basis for the repository. This meta-meta-model is instantiated to reflect a given meta-model, which is then further instantiated when mapping the internal data of its tool to the information model of the repository.

We adopt a simple meta-meta-model which generalises among established meta-meta-modelling languages such as MoF [12], Dome [13] and GME [14], and based on a broad survey of modelling languages for embedded computer systems [15]. A model can be generally viewed as consisting of a hierarchical structuring of modelling objects that may possess properties; ports defining interfaces of these objects; and relationships (such as associations, inheritance and refinement) between ports. Modelling languages differ in the kinds of objects that can be specified, their relationships and the kind of properties they possess. When integrating a particular model, a meta-model is instantiated by defining the kind of objects, ports and relations that exist in a model. (Note that the main aim is not to suggest yet another meta-meta-model that claims to cover any modelling language. A simple, generalised meta-meta-model was adopted, allowing focus to be placed on the PDM/SCM integration aspects of the platform.)

Figure 2 shows a UML class diagram of the object types, attributes and relations defining the generic meta-meta-model. As an example, the lower part of the figure illustrates the meta-model of a Data Flow Diagram (DFD) [21] model as interpreted by the Simulink tool [11], which is defined by specialising the generic objects.

In this approach, the granularity at which the MDM system operates on the models is controlled by the definition of the meta-model, implemented in the adaption layer. MDM mechanisms will understand the model semantics down to the level at which the elements, ports and properties are defined. Finer semantics within these entities are not the concern of MDM. For example, if a property of an element is defined as a blob of text, an MDM functionality cannot be expected to interpret the detailed semantics of this property.

Adopting a common meta-meta-model between the models is not sufficient if there is a need to integrate the various model contents into a whole. For this to be possible, a unified information model of the set of models is necessary, specifying more detailed semantics of the models and their interrelations. While such information models are standardised for hardware products [16], no such standard model is currently available that also encompasses models from the software discipline. In [17], ongoing work on how such integration can be achieved is presented.

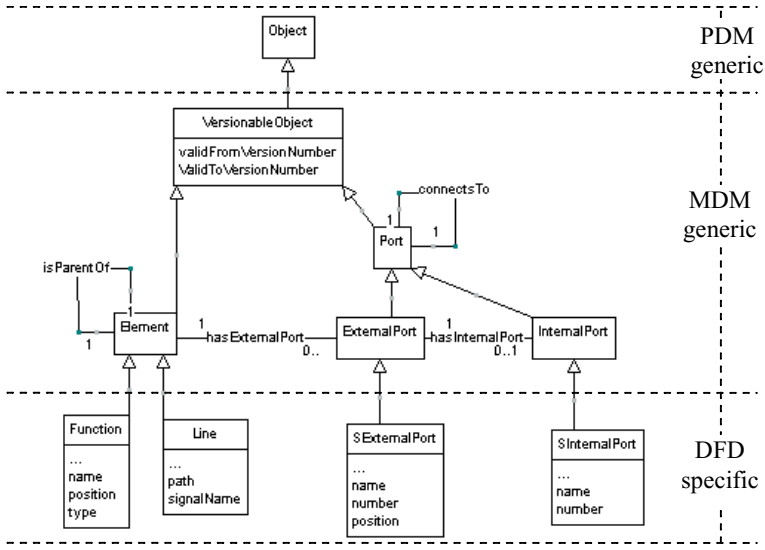


Fig. 2. The PDM information model implementing the meta-meta model, and showing how a tool-specific meta-model is defined.

Tool interface Access to the tool data and the mapping of this data to the repository is performed by an *adaption layer*. An adaption layer is developed for each tool to be integrated into the MDM system. This layer isolates the tool-specific issues allowing MDM to operate generically on many tools implementing different technologies. The adaption layer fulfils three purposes. As discussed in the previous section, it maps the specific meta-model of its designated tool onto the repository.

Second, the adaption layer maps the tool-specific format used internally to manage the model data to a generic format of the repository. In this way, the management functionalities can operate uniformly using a single format.

Different technologies are available for a tool to internally store its model data. A tool can use either a computer file system to store model data in a file, or a database management system. Various standards exist that specify how data should be handled using these technologies, yet one cannot assume that tools will not implement their own solutions.

In a set of tools in which the tools adopt a combination of technologies (standard or not), it becomes necessary to translate these technologies onto a common format, in order to make the interface to the MDM platform generic. This role is fulfilled by the adaption layer, making the tool-specific data technology transparent to the rest of the platform. The adaption layer translates the format used by its designated tool to the chosen format of the repository. In the platform advocated in this paper, we adopt the data neutral XML format to interface the adaption layer to the repository.

Third, the adaption layer accommodates the different techniques used to gain access to the tool's internal data. Different tools use different technologies to provide automated access to its internal data. In the simplest case, the adaption layer can ac-

cess and interpret the textual file produced by the tool. A tool can also provide 'export' functionality, an Application Programming Interface (API), or a query language.

For a potential tool to be integrated into the MDM system, specific automation support is expected. In order to allow fine-grained accessibility to parts of models and the manipulation of models, a modelling tool whose models are to be managed need to:

- Provide access to the model data either through an API or using parsable text [16].
- Provide fine-grained mechanisms for the construction and modification of models through an API.

Again, in this a-posteriori approach, an integrated tool needs neither to conform to the meta-meta-model, nor format, nor data access approach adopted by the MDM platform. Such demands would create a tight, undesired, dependency between the integrated tools and the platform. It is the adaption layer's role to map these technologies to those of the platform.

Management Functionalities MDM functionalities ought to generically store and handle models from the various tools and disciplines. The functionalities of the union of typical SCM and PDM tools would include: Version management, product structure management, build management, change management, release management, workflow and process management, document management, concurrent development, configuration management and workspace management [4].

The model-based approach to data management unifies the disciplines by unifying the kinds of objects it manages – models. The management functionalities should focus on the models and their contents, transparent of the file structure used to store them.

While as much of the functionalities can be shared by the disciplines, discipline-specific functionalities need still to be supported such as build management for the software discipline. In certain cases, it may also be desired to provide different solutions of the same functionality for different disciplines. For example, software development might require the complex version control mechanisms and concurrent development normally provided by SCM systems, while hardware development is satisfied with sequential revision control. There should be no problem providing different solutions in MDM, depending on the kind of data items the functionality operates on. It would however be advantageous to base the different solutions on generic mechanisms for reusability purposes. The different solutions ought to be also based on the same user interface and terminology.

In order to test the proposed architecture, we have investigated in details the version management functionality of models. This functionality is termed Model Version Control (MVC). While version control is needed in both domains, the functionality differs between SCM and PDM systems (section 2.2). This allows us to investigate how far such mechanisms can be aligned between the disciplines. Version control is also critical since it will put to the test the other crucial factors discussed in section 2, such as the possibility of having a common product structure and data representation. A short summary of MVC is presented in section 3.4.

3.4 Model Version Control Implementation

The MDM platform is currently developed using the Matrix PDM system [19]. As shown in figure 2, it was necessary to specialise the meta-meta model provided by the tool in order to perform the desired version control algorithm.

A simple model version control functionality (MVC) has been implemented. This should be seen as an exploration of the integration possibilities of model-based development. The implementation borrows from the general ideas from the fine-grained version control algorithms such as [5] and [20]. However, instead of using conventional databases, we base our implementation on the MDM architecture.

The algorithm supports the versioning of any model that can be mapped to the meta-meta-model assumed in the platform. In the current implementation, data flow diagram (DFD) [21] models from the Matlab/Simulink [11] tool and Hardware Structure Diagram models [22] in the Dome [13] tool, are handled.

Even though each tool's models contain a different kind of modelling objects, with different set of properties, MVC operates generically on all kinds of models, owing to the adoption layer which presents the model instances using a common format and structure.

Compared to version control mechanisms in conventional SCM systems, the major difference with the model-based approach is that entities have relations between them that also need to be handled. Such relations do not exist between files in the file-based approach. In file-based version control, the versioning of an entity (file) is done independently, and does not affect the versioning of other entities in the system, since no relations exist between them. In contrast, the versionable objects of a model are interrelated and creating a new version of an object might influence the versions of others.

Similar to file-based SCM systems, by only saving the changes between versions of a model, this algorithm maintains efficient storage in the repository and avoids the duplication of information. In addition, comparison between different versions of a model can be efficiently deduced.

MVC provides mechanisms that allow a user to save and extract any part of the system model. In a 'checkin' operation, changes to the model since the last checkin operation are saved in the repository. When performing a 'checkout' operation, the specified element is reconstructed for a given version, together with its subparts, forming an XML document of the information in the repository. This document is then further transformed by the adaption layer to create a tool-specific format that can be used by the tool. The details of these operations are performed transparently to the user, allowing him/her to interface with the modelling tool's interface and format. Further details on the implemented algorithm can be found in [18].

3.5 Integration vs. Unification

As an alternative to integrating PDM and SCM systems as proposed in [4] and [6], the MDM architecture ought to be interpreted as a unified solution that aims to support the needs of both disciplines, assuming model-based development.

The need to move from the file-based approach of SCM to focus on models instead, makes much of the mechanisms currently available technically obsolete. So, the

only advantage of maintaining both systems using the integration approach would be to maintain the user interface and terminologies software engineers are accustomed to. Integration techniques struggle with trying to synchronise and balance between the two disciplines.

Instead, the unification approach imposes new common mechanisms with common terminology that are expected to be accepted by both disciplines. Naturally, this approach faces more resistance from established developers and disciplines. However, the shift to model-based development would require a paradigm change that the software community may have to face anyway.

Failures in PDM/SCM integration efforts due to cultural differences [1] ought to be seen as integration problems in the organisation itself that have to be dealt with. In the best case, a unified approach can only bring the conflicts to the surface to be dealt with appropriately.

Accepting the resistance and time it takes tool vendors to change, integration may be the first step, but the future is unification.

4 Related Work

SCM systems targeting models, instead of file objects, are increasingly appearing in the literature ([5], [20] and [23]). In these approaches, an information model of the documents to be handled is assumed, allowing for the management of the internal information stored in the documents, as well as the specification of relations between information from different documents. While focused on software models, these approaches are helpful since the mechanisms can be extended to apply to any kind of models throughout the development lifecycle. The MVC implementation advocated in this paper is inspired by these approaches, broadening their use for more general model types. More importantly, basing the implementation on the facilities already available in PDM systems, instead of using conventional databases, helps in the integration with the mechanisms in the discipline of hardware development.

In [4], three techniques of integrating PDM and SCM systems are proposed. Of these, the full integration technique was considered ideal and most desired. In the full integration solution, the systems' functionalities are separated from their own repositories, and reintegrated into a common repository with a common information model. A common user interface is also built on top, in order to give all users a common look and feel. However, it is argued that full integration is difficult to implement using today's tools due to the tight integration of the tools' components. All the suggested approaches accept the status quo of software and hardware development and consequently needed to deal with fundamental differences. This lead to limited integration success. Rejecting the status quo and focusing on the commonality between the disciplines (model-based development), should instead lead to a smoother integration.

In [6], a configuration management system is suggested that can be applied to both software and hardware design documents. The system also allows for relationships, such as dependencies, to be established between documents. However, the entities handled by the system are file documents with no fine-grained management of their content.

5 Conclusion

In multidisciplinary development, the integration of the various management systems used by different disciplines is of critical value. An integrated environment allows the efforts of all developers to be well communicated and reduces any risks of inconsistencies and conflicts between them.

Due to the difference in maturity levels of these disciplines, such integration efforts has had limited success in the past. Specifically, the implementation-centred development approach of software systems expected a coarse-grained support from SCM systems, where documents are the smallest entities managed, while ignoring the internal model semantics contained within them. In comparison, mechanical development expects the handling of the detailed product data by their corresponding PDM systems using standard information models.

However, with the move towards model-based development, where the use of models becomes the central activity in making, communicating and documenting design decisions, disciplines share a common need to handle the same kind of entities – models. In this way, management systems can be brought closer together.

This paper presented an architecture for a Model Data Management (MDM) system that aims to provide the support functionalities expected of a model-based development environment. The system aims to generically handle and control the various kinds of models produced by the different tools during the development of software-intensive, yet multidisciplinary, products. The proposed architecture builds on existing technologies from the more mature discipline of mechanical engineering, while borrowing new ideas from the software engineering discipline.

To illustrate the MDM solution, an initial implementation of a Model Version Control (MVC) functionality was performed, allowing for the fine-grained version management of two types of models from two different tools. MVC permits stakeholders to perform design activities in terms of models, where they can organise, share and modify their models, transparent to the underlying file structure. A simplified version control functionality has been realised. The ability to perform branches and merges in the changes of an element is a very important feature of version control, specifically desired in software development. This is needed in order to study different design alternative, provide product variants, or deal with a bug fix from an earlier release. MVC needs to handle this functionality in the future.

The major aim of the current platform implementation has been to experiment and illustrate the concepts discussed in this paper. While the current implementation has only been validated through the use of a small case study, a more commercial size case study would be needed to appropriately validate the usability of this approach. This remains to be done in the future.

The advantage of MDM over conventional PDM/SCM systems is the inclusion of the internal content of its supported models, allowing for a tighter integration of the design information between different models. In addition, functionalities are generically applicable for many kinds of models, simplifying the process of adding new tools into the toolset. However, an initial effort is required to integrate new models in the development of the adaption layer. The fine-grained management of models is bound to require more computational effort that the coarse-grained approach.

The development process of software and hardware products will always differ due to the nature of the products themselves. However, in a unified approach the same mechanisms ought to be used to support these differing processes. Moreover, by providing different strategies for different kinds of models, the development needs of both disciplines can be satisfied, using variants of the same basic mechanisms in a unified management system. It is essential however to base the strategies on the same basic mechanisms and user interface, allowing the reuse of basic components and preventing confusion in terminologies.

In the case where development is not (completely) model-based, MDM facilities may still be used. Any product data inputted into the platform is restructured and interpreted to form model data. For example, a MDM system can manage the files of a Java project by reinterpreting each file as a class model, extracting and managing the attributes and methods contained within each file as fine-grained structured data.

The approach is currently implemented using a PDM system. It is our ideal vision that with the acceptance of model-based development, one no longer needs to discuss the integration of PDM and SCM systems. Instead, a truly unified approach to model data management can be used by both disciplines.

6 Acknowledgements

This work has been supported by the Swedish Strategic Research Foundation, through the SAVE project.

References

1. Dahlqvist, A.P., Crnkovic, I. and Asklund, U., Quality Improvements by Integrating Development Processes, 11th Asia-Pacific Software Engineering Conference, 2004.
2. OMG, Model Driven Architecture Specification, MDA Guide Version 1.0.1, Document Number: omg/2003-06-01, June 2003.
3. OMG, Unified Modeling Language (UML) Specification, V1.5, March 2003.
4. Crnkovic I., Asklund U. and Persson Dahlqvist A., Implementing and integrating product data management and software configuration management, Artech House Publishers, 2003.
5. Ohst D. and Kelter U., A fine-grained version and configuration model in analysis and design, Proceedings of the International Conference on Software Maintenance, 2002.
6. Westfechtel B. and Conradi R., "Software Configuration Management and Engineering Data Management: Differences and Similarities" Proceedings 8th International Workshop on System Configuration Management, Springer-Verlag, pages 95-106, 1998.
7. Kemmerer S. J. (editor), "STEP, the grand experience", National Institute of Standards and Technology, special publication 939, 1999.
8. Estublier J., Favre J. M. and Morat P., Toward SCM / PDM integration?, International Workshop on Software Configuration Management, (SCM8), Springer Verlag, 1998.
9. Kruchten, P., Casting Software Design in the Function-Behavior-Structure Framework, IEEE Software, Volume 22, Issue 2, 2005.
10. Ohst D., Welle M. and Kelter U., "Differences between Versions of UML Diagrams", Proceedings of the joint European software engineering conference (ESEC) and SIGSOFT symposium on the foundations of software engineering (FSE-11), 2003.

11. Simulink, Mathworks, http://www.mathworks.com/products/simulink/, accessed March 2005.
12. OMG, Meta Object Facility (MOF) Specification, V1.4, April 2002.
13. Dome, "Dome Guide" Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999.
14. GME, A Generic Modeling Environment, GME 4 User's Manual, Version 4.0, Institute for Software Integrated Systems, Vanderbilt University, 2004.
15. El-khoury J., Chen D. and Törngren M., "A survey of modelling approaches for embedded computer control systems (Version 2.0)" Technical report, ISRN/KTH/MMK/R-03/11-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.
16. Kemmerer S. J. (editor), STEP, the grand experience, National Institute of Standards and Technology, special publication 939, 1999.
17. El-khoury J., Redell O. and Törngren M., A Tool Integration Platform for Multi-Disciplinary Development, to be published, 31st Euromicro Conference on Software Engineering and Advanced Applications, 2005.
18. El-khoury J and Redell O., A Model Data Management Architecture for Multidisciplinary Development, Internal Technical Report, Mechatronics Lab. Royal Institute of Technology, Stockholm. 2005.
19. MatrixOne, Matrix10, http://www.matrixone.com/, accessed April 2005.
20. Nguyen T. N., Munson E.V., Boyland J.T. and Thao C., Flexible Fine-grained Version Control for Software Documents, 11th Asia-Pacific Software Engineering Conference, 2004.
21. Cooling J., Software Engineering for Real-time Systems. Pearson Education Limited, ISBN 0201596202, 2003.
22. Redell O., El-khoury J. and Törngren M., The AIDA toolset for design and implementation analysis of distributed real-time control systems, Microprocessors and Microsystems, Volume 28, Issue 4, 2004.
23. Chien S. Y., Tsotras V. J., Zaniolo C., Version Management of XML Documents, Third International Workshop WebDB 2000 on The World Wide Web and Databases, 2000.
24. Becker S. M., Haase T. and Westfechtel B., Model-based a-posteriori integration of engineering tools for incremental development processes, Journal of Software and Systems Modeling, Volume 4, Number 2, Springer, 2005.

Observations on Versioning of Off-the-Shelf Components in Industrial Projects (short paper)

Reidar Conradi[1,2] and Jingyue Li[1]

[1]Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
[2]Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
{conradi, jingyue}@idi.ntnu.no

Abstract. Using OTS (Off-The-Shelf) components in software projects has become increasing popular in the IT industry. After project managers opt for OTS components, they can decide to use COTS (Commercial-Off-The-Shelf) components or OSS (Open Source Software) components. We have done a series of interviews and surveys to document and understand industrial practice with OTS-based development in Norwegian, German, and Italian IT industry. The perspective is that of a software architect or system integrator, not a developer or maintainer of such components. The study object is a completed development project using one or several OTS components. This paper reports on the versioning aspects of OTS components in such projects. We found that one third of the COTS components actually provided source code, in addition to all OSS components. However, OTS components were seldom modified (i.e. reused "as-is"), even if source code was available. Although backward compatibility of new releases did not cause noticeable problems for a single OTS component, using several different OTS components in a project caused difficulties in maintenance planning of asynchronous releases and system integration of new releases. Several new research questions have been formulated based on the results of this study.

1. Introduction

Software reuse in the form of *component-based software development (CBSD)* has long been proposed as a "silver bullet". It is supposed to offer lower cost, shorter time-to-market, higher quality, and stricter adherence to software standards. Software developers are therefore increasingly using COTS (Commercial-Off-The-Shelf) and OSS (Open Source Software) components in their projects, commonly called OTS (Off-the Shelf) components. COTS components are owned by commercial vendors, and their users normally do not have access to the source code [1]. On the other hand, OSS components are provided by open source communities, with full access to the source code [6].

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

The granularity of an OTS component can be different. Some regard that OTS components could or should include very large software packages such as Microsoft Office. Others limit OTS components to GUI libraries. In this study, we focus on OTS components as *software components*. Such a component is a unit of composition, and must be specified so that it can be composed with other components and integrated into a system (product) in a predictable way [10]. That is, a component is an "*Executable unit of independent production, acquisition, and deployment that can be composed into a functioning system.*" We also limit ourselves to components that have been explicitly decided either to be built from scratch or to be acquired externally as an OTS-component. That is, to components that are not shipped with the operating system, not provided by the development environment, and not included in any pre-existing platform. That is, platform ("commodity") softwares are not considered, e.g. an OS like Linux, DBMSes, various servers, or similar softwares. This definition implies that we include not only components following COM, CORBA, and EJB standards, but also software libraries like those in C++ or Java. This definition is consistent with the scope used in the component marketplace [9].

To record, understand, and possibly improve industrial practice wrt. OTS-based development, we have carried out several empirical studies of on the usage of COTS and OSS components. This paper will report some results from these studies wrt. versioning of such components. The remainder of this paper is organized as follows: Section two motivates and states the research questions and the research method. Section three describes the results and section four discusses these. Finally, conclusions and future research are presented in section five.

2. Research questions, research method, and data collection

2.1 Motivation and some context

There is a growing literature on OTS-based development, but alas with few representative studies on industrial practice. For instance, Torchiano and Morisio [7] interviewed 7 small IT companies in Norway and Italy on their experience with COTS-based development. Even by this tiny study, they stated six theses that refute many assumptions from the literature. For instance, they claim that OSS and COTS components are used very much in a similar way, e.g. that components are normally not modified even if source code is available.

Based on their study, we first performed a pre-study on COTS components as structured interviews of 16 COTS-based projects in 13 Norwegian IT companies [4]. From the pre-study, we gathered some new insights on COTS-based development and clarified our research questions and hypotheses. The study presented in this paper extended the pre-study in two dimensions. First, it included OSS components because they represent an alternative to COTS components. Second, this study included samples from Norway, Italy and Germany. In addition, the sample was selected randomly instead of on convenience as in the pre-study. The study was performed as a survey

with a web-questionnaire, using a randomized sample of 133 projects from small, medium, and large IT companies [2, 5]. The perspective was largely that of a system integrator.

2.2 Research questions

To investigate the state-of-the-practice of versioning problems in OTS-based development, we first designed research question RQ1 to study separate OTS components. We then designed research question RQ2 to study the whole OTS-based project, which might use several different OTS components.

2.2.1 Research question RQ1: In RQ1, we want to know whether and to what extent OTS components are actually modified locally. The source code is namely available not only for OSS components, but also for many COTS components [4]. But even if the need and opportunity is there, will such changes actually be performed? The first research question is therefore:

RQ1: *To what extent are OTS components actually modified locally?*

2.2.2 Research question RQ2: Some OTS-based projects integrate several OTS components. Updates (releases) to these components may contain unpredictable functionality and come at different intervals − on which a system integrator has little control. Previous studies indicate that the number of different OTS components used in one project has a strong relationship with maintenance effort [1, 3], even postulates that maintenance costs depend on the square of the number of components. The second research question is:

RQ2: *Is system maintenance perceived to carry a risk due to future versioning incompatibilities?*

2.3 Research design

To clarify **RQ1** and **RQ2** we will use data from the mentioned survey [5]. However, the survey questions were not designed to collect comprehensive data on versioning-related issues. We have selected the survey questions given below, with boldfacing as in the original questionnaire.

For **RQ1**, we first asked the respondent to select the most important OTS component in their project, i.e. providing the most functionality for the actual application. This is named **Comp.1** below. Then we asked questions Q5.4, Q5.5, Q5.6 and Q5.10.

- Q5.4: What was the **source code status** of the selected component **Comp.1**? The options are:
  - OSS, i.e. with source code available
  - COTS, but with source code available
  - COTS, without source code
- Q5.5: Have you **read parts of** the source code of OTS-component **Comp.1**? We used a five-point Likert scale (very little, little, some, much, very much -

plus don't know) to measure the answers to this question. The answers were mapped to ordinal values 1 to 5 later.

- Q5.6: Have you **modified parts of** the source code of OTS-component **Comp.1**? We used the same measurement scale as in Q5.5.
- Q5.10 Did you encounter some of the following **aspects (risks)** with the selected OTS component **Comp.1**? The relevant option is:
  - Q5.10.d: The recent OTS component **versions** were **not backward-compatible** with the pervious version.

  We used only yes, no, and don't know to measure the answer to this question.

For **RQ2**, we asked about possible versioning-maintenance problems of the whole project through question Q4.1 and three sub-questions:

- Q4.1 What is your opinion on the following **aspects (risks)** of your OTS-based project? The relevant sub-questions are:
  - Q4.1.k: It was difficult to **plan** system **maintenance**, e.g. because different OTS components had asynchronous release cycles.
  - Q4.1.l: It was difficult to **update the system** with the **last OTS component version**.
  - Q4.1.m: OTS components were not satisfactorily **compatible with the production environment** when the system was deployed.

For each sub-question, we used another five-point Likert scale (don't agree at all, hardly agree, agree somewhat, agree mostly, strongly agree - and don't know) to measure the answer. The answers were mapped to ordinal values 1 to 5 later.

For **RQ2**, we also investigated whether the number of different OTS components used in the project will influence integration effort as measured in Q4.1. We gathered the relevant information through question Q5.1: How many different OTS components have been used in the project?

2.4 Data collection and analysis

The unit of study for **RQ1** and **RQ2** is a completed software development project. Sampling is described elsewhere [2], and data was mostly collected via a web-tool. According to the focus of the different research questions, we used different data analysis methods:

- For **RQ1,** we first clustered OTS components into two categories, with source code and without source code, according to the answers of Q5.4. We then analyzed the distribution of answers to questions Q5.5 and Q5.6 concerning OTS components with source code. After that, we calculate the distribution of answers to question Q5.10.
- For **RQ2**, we first studied the distribution of answers to Q4.1. We then calculate the correlation between the possible risks (Q4.1.k, Q4.1.l and Q4.1.m) with the number of different OTS components used in the project (Q5.1).

3. Research results

We have gathered results from 133 projects (47 from Norway, 48 from Germany, and 38 from Italy). Three companies gave results for more than one project. In these 133 projects, 83 used only COTS components, 44 used only OSS components, and six used both COTS and OSS components. For these six projects, five of them gave detailed information of one COTS component, and one gave information of an OSS component. In total, we gathered detailed information on 88 COTS components and 45 OSS components.

3.1 Answers to research question RQ1

For **RQ1**, the answers to Q5.4 show that 29 (or 1/3) of 88 COTS components actually made available the source code to their users, i.e. software integrators.

The general distribution of answers to Q5.5 is shown in Figure 1. It shows that the median value concerning reading of COTS components is 3 (meaning some). The median value of OSS components is the same. This means that 1/3 of the COTS components (having available source code) and all the OSS components are *read* to some degree.

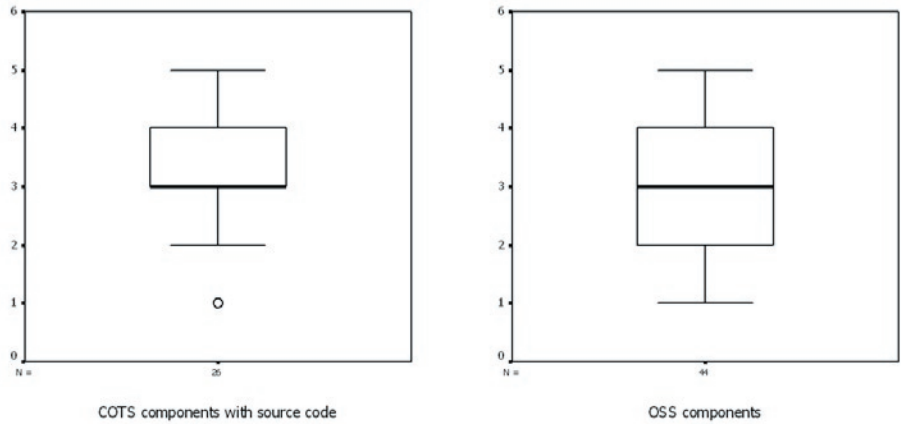


Figure 1. Answers to Q5.5: Has the source code been read?

The detailed answers to question Q5.5 is shown in Table 1. We observe that COTS components with source code were actually read slightly more frequently than their OSS counterparts.

Table 1. Detailed answers of Q5.5 (source code reading)

	Valid answers	**Read somewhat (with value more than 3)**
COTS components with source code	26 out of 29	20 out of 26 (77%)
OSS components	44 out of 45	30 out of 44 (68%)

Figure 2 and Table 2 below shows similarly the answers to Q5.6. Figure 2 shows that the COTS components with source code have been somewhat *modified,* i.e. with a median value of 2 (meaning little). OSS components – all with source code – had also been somewhat modified and with the same median value. The distribution indicates that users *less frequently modify than read* the source code of OTS components, even it such source was available. In Table 1 above, we saw that COTS components with source code were *more* frequently *read* than their OSS counterparts. Table 2 shows that OSS components were *more* frequently *modified* than their COTS counterparts. In the pre-study [4], respondents often expressed that they wanted to perform certain source code modifications of a component, but decided not to perform these for fear of costly maintenance and re-integration with future releases.

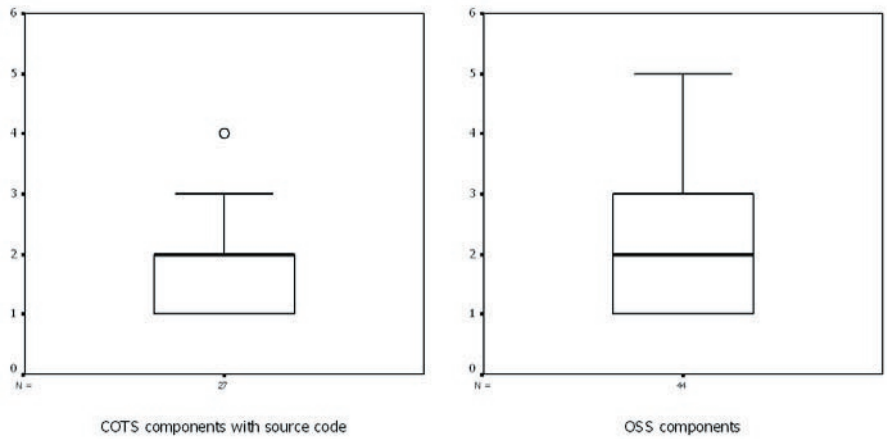


Figure 2. Answers to Q5.6: Has the source code been modified?

Table 2. Detailed answers of Q5.6 (source code modification)

	Valid answers	**Modified somewhat (with value more than 3)**
COTS components with source code	27 out of 29	4 out of 27 (15%)
OSS components	44 out of 45	16 out of 44 (36%)

The result of Q5.10 (backward compatibility) is finally shown in Table 3. The results show that only 17% (10 out of 59) of COTS components and 11% (3 out of 27) of OSS components had back compatibility problems. From this we can conclude that versioning-maintenance problems were not frequent in the selected OTS components. It also shows that there is no significant difference of backward compatibility problems between COTS and OSS components.

Table 3. Result of Q5.10 on backward compatibility problems.

	Yes	No	Don't know	All (N)
COTS component	10	59	19	88
OSS component	3	27	15	45

3.2. Answers to research question RQ2

For **RQ2**, the answers of sub-questions Q4.1.k (asynchronous release cycles), Q4.1.l (last version gives system update problems), and Q4.1.m (last version gives problems with production environment) are shown in Figure 3. We do not tell the differences between projects using COTS and OSS because they are facing the same versioning risk, i.e. that OTS component versioning is out of the OTS component users' control.

Results of Q4.1 show that the median values of Q4.1.k, Q4.1.l, and Q4.1.m are all 2 (meaning hardly agree). It means most OTS component users did not regard the versioning mismatches as a serious maintenance risk in general.

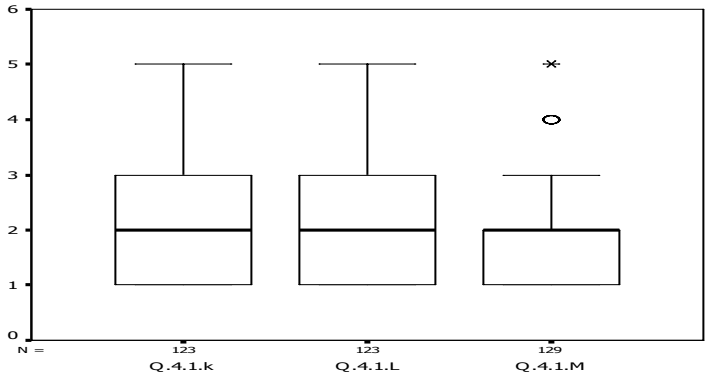


Figure 3. Answers to Q4.1 on some project versioning risks.

To investigate the correlation between the above versioning risks and the number of different OTS components in one project, we used *Spearman* rank correlations in SPSS 11.0. Although the number of different OTS components is an interval (integer) variable, we used it as an ordinal variable. That is, we gave a project using less OTS component a lower rank than a project using more OTS components. The relationship between answers to Q4.1.k, Q4.1.l, and Q4.1.m and the number of different OTS components is shown in Table 4.

Table 4. Correlation between "versioning problems" and number of different OTS components

	Correlation coefficient	Significance (2-tailed)
4.1.k with number	.182	.047*
4.1.l with number	.289	.001*
4.1.m with number	.027	.760

*Correlation is significant at the .05 level (2-tailed).

From Table 4, we can see that the number of different OTS components used in the project have a significant effect on the asynchronous release-cycle problem (Q4.1.k), on the last-version-gives-system-update problem (Q4.1.l).

4. Discussion

This study is basically a state-of-the-practice survey, where we observed some basic trends in OTS-based development in industry. These observations invoked several new research questions that we would like to investigate in the future.

4.1 New Research Question NRQ1: Why OTS source code was seldom modified?

For this study, we discovered that most OTS component users do read the source code when it is available. However, OTS component users did not change the source code very much. Some studies assume that users didn't need to see or modify, or lacked the knowledge, skills or resources to do so [7]. Another possible reason is that the users fear costly future maintenance (cost of reintegration) when a new OTS component version is released [4]. If the reason is the latter one, a new research question **NRQ1** will be: *If it is necessary to locally modify the source code in an OTS component, how to support integration of new OTS component versions (releases) with the local modifications?* An obvious remedy is to apply (semi-)automatic merge tools, often as part of common SCM tools. In the OSS community there is heavy use of OSS's own bug tracking tool, Bugzilla [8] that again uses the open CVS tool for versioning. Furthermore, there is a commitment in the OSS community to *report back* local modifications. Thus the merge/integration job may possibly be delegated to the "owner" of an OSS component. However, we have no specific information in this survey on such integration or any use of SCM tools.

4.2 New Research Question NRQ2: How to manage versioning problems when using several OTS components in the same project?

Results of Q5.10 show that the versioning problems of reusing a single OTS component are very few. However, the versioning risk will increase as the number of different OTS components increases. Our data gives further support the findings in [3], that the most significant factor that influences lifecycle cost of a COTS-based system is the number of COTS packages that must be synchronized within a release.

However, our study shows that using more than one OTS component in a project is sometimes unavoidable. 90 of the 133 projects used more than one OTS component. Therefore, another interesting research question **NRQ2** is: *How to estimate the "optimal" number of OTS components in a project to balance initial development savings with later maintenance costs?* Moreover, some of our investigated projects had very few versioning problems, even if they used more than 10 different OTS compo-

nents in their project. Summarizing their experience by case studies to give guidelines on OTS- based development could be yet another, new research question.

4.3 Possible treats to validity

Construct validity In this study, most variables and alternatives are taken directly, or with little modification, from existing literature. The questionnaire was pre-tested using a paper version by 10 internal experts and 8 industrial respondents before being published on a web tool. About 15% of the questions have been revised based on pre-test results. However, a possible threat to construct validity is that we forgot to give a clear "no" alternative in questions Q5.5 and Q5.6 (not only "very little", "little" etc.).

Internal validity We promised respondents in this study a final report and a seminar to share experience. The respondents were typically persons who wanted to share their experience and wanted to learn from others. We therefore think that the respondents answered the questionnaire truthfully. However, different persons in the same project might have different opinions on the same project. Asking only one person in each project might not be able to reveal the whole picture of the project. Due to length limitation of a questionnaire, we asked the respondent to fill in information for only one component in the project. The possible threat is that other OTS components in the same project might lead to different answers to our questions.

Conclusion validity This study is a state-of-the-practice study. We studied what had happened in industrial projects. However, we did not investigate the cause-effect relation of the phenomena discovered in this study. The sample size is generally sufficient for valid statistical conclusions.

External validity We used different randomization to select samples in different countries. However, the sample selection processes were not exactly the same due to resource limitations [2]. Another possible threat to external validity is that our study focused on fine-grained OTS components. Conclusions may be different in projects using complex and large OTS packages, such as ERP, content management systems, and web services in general.

5. Conclusion and future work

This paper has presented results of a state-of-the practice survey on OTS-based development in industrial projects. The results of this study have answered two questions relevant for software configuration management:

- **RQ1**: *To what extent are OTS components actually modified locally?*

Our results show that most OTS component users took advantage of the available source code and read it. However, few of them actually modified it.

- **RQ2**: *Is system maintenance perceived to carry a risk due to future versioning incompatibilities?*

Our results show that versioning problems when using a single OTS component were few. However, the key challenge is to coordinate versioning when several OTS components were used in the project.

Results of this study have shown state-of-the-practice data. By observing the current trend in industry, we discovered several interesting research questions to be studied in the future. The next step is to do a larger qualitative study with personal interviews to further study some of the new research questions.

Acknowledgements

This study was partially funded by the INCO (INcremental COmponent based development, http://www.ifi.uio.no/~isu/INCO) project. We thank the colleagues in this project, and all the participants in the survey. We also thank the local OSS enthusiast Thomas Østerlie for valuable comments.

References

1. Basili, V. R. and Boehm, B.: COTS-Based Systems Top 10 List. IEEE Computer, 34(5):91-93, May/June 2001.
2. Conradi, R., Li, J., Slyngstad, O. P. N., Bunse, C., Kampenes, V.B., Torchiano, M., and Morisio, M.: Reflections on conducting an international CBSE survey in ICT industry. Submitted to 4th International Symposium on Empirical Software Engineering (ISESE'05), 17-18 Nov. 2005, Noosa Heads, Australia, 11 pages.
3. Donald, J. R., Basili, V., Boehm, B., and Clark, B.: Eight Lessons Learned during COTS-Based Systems Maintenance. IEEE Software, 20(5):94-96, Sep./Oct. 2003.
4. Li, J., Bjørnson, F. O., Conradi, R., and Kampenes, V. B.: An Empirical Study of Variations in COTS-based Software Development Processes in Norwegian IT Industry. Submitted to the Journal of Empirical Software Engineering, 29 pages.
5. Li, J., Conradi, R., Slyngstad, O. P. N., Bunse, C., Khan, U., Torchiano, M., and Morisio, M.: An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects. Proc. 6th International Conference on Product Focused Software Process Improvement (PROFES'2005), 13-16 June, 2005, Oulu, Finland, Springer Verlag LNCS Volume 3547, pp. 54 - 68
6. Open Source Initiative (2004): http://www.opensource.org/index.php
7. Torchiano, M. and Morisio, M.: Overlooked Aspects of COTS-based Develop*ment.* IEEE Software, 21(2):88-93, March/April 2004.
8. Bugzilla tool used for OSS (2005): http://www.bugzilla.org/
9. ComponentSource (2004): http://www.componentsource.com/
10. Crnkovic, I., Hnich, B., Jonsson, T., and Kiziltan, Z.: Specification, Implementation, and Deployment of Components. Communication of the ACM, 45(10):35 – 40, October 2002.

Continuous Release and Upgrade of Component-Based Software*

Tijs van der Storm

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands, storm@cwi.nl

Abstract. We show how under certain assumptions, the release and delivery of software updates can be automated in the context of component-based systems. These updates allow features or fixes to be delivered to users more quickly. Furthermore, user feedback is more accurate, thus enabling quicker response to defects encountered in the field.
Based on a formal product model we extend the process of continuous integration to enable the agile and automatic release of software components component. From such releases traceable and incremental updates are derived.
We have validated our solution with a prototype tool that computes and delivers updates for a component-based software system developed at CWI.

1 Introduction

Software vendors are interested in delivering bug-free software to their customers as soon as possible. Recently, *ACM Queue* devoted an issue to update management. This can be seen as a sign of an increased awareness that software updates can be a major competitive advantage. Moreover, the editorial of the issue [7], raised the question of how to deliver updates in a component-based fashion. This way, users only get the features they require and they do not have to engage in obtaining large, monolithic, destabilizing updates.

We present and analyse a technique to automatically produce updates for component-based systems from build and testing processes. Based on knowledge extracted from these processes and formal reasoning it is possible to generate incremental updates.

Updates are produced on a per-component basis. They contain fine-grained bills of materials, recording version information and dependency information. Users are free to choose whether they accept an upgrade or not within the bounds of consistency. They can be up-to-date at any time without additional overhead from development. Moreover, continuous upgrading enables continuous user feedback, allowing development to respond more quickly to software bugs.

The contributions of this paper are:

* This work was sponsored in part by the Netherlands Organisation for Scientific Research, NWO, Jacquard project DELIVER.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

- An analysis of the technical aspects of component-based release and update management.
- The formalisation of this problem domain using the relational calculus. The result is a formal, versioned product model [4].
- The design of a continuous release and update system based on this formalisation

The organisation of this paper is as follows. In Section 2 we will elaborate on the problem domain. The concepts of continuous release and upgrade are motivated and we give an overview of our solution. Section 3 presents the formalisation of continuous integration and continuous release in the form of a versioned product model. It will be used in the subsequent section to derive continuous updates (Section 4). Section 5 discusses the prototype tool that we have developed to validate the product model in practice. In Section 6 we discuss links to related work. Finally, we present a conclusion and list directions for future work in Section 7.

2 Problem Statement

2.1 Motivation

Component-based releasing presumes that a component can be released only if its dependencies are released [18]. Often, the version number of a released component and its dependencies are specified in some file (such as an RPM spec file [1]). If a component is released, the declaration of its version number is updated, as well as the declaration of its dependencies, since such dependencies always refer to released components as well. This makes component-based releasing a recursive process.

There is an substantial cost associated with this way of releasing. The more often a dependent component is released, the more often components depending on it should be released to take advantage of the additional quality of functionality contained in it. Furthermore, on every release of a dependency, all components that use it should be integration tested with it, before they can be released themselves.

We have observed that in practice the tendency is to not release components in a component-based way, but instead release all components at once when the largest composition is scheduled to be released. So instead of releasing each component independently, as suggested by the independent evolution history of each component, there implicitly exists a practice of big-bang releasing (which inherits all the perils of big-bang integration[1]).

One could argue, that such big-bang releases go against the philosophy of component-based development. If all components are released at once as part of a whole (the system or application), then it is unlikely that there ever are two components that depend on different versions of the same component. Version

[1] See `http://c2.com/cgi/wiki?IntegrationHell` for a discussion.

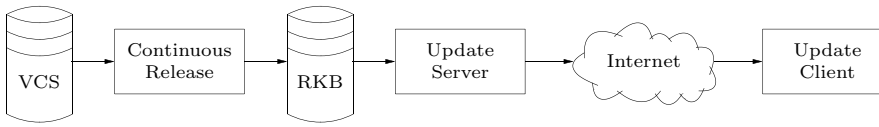


Fig. 1. Continuous Release Architecture

numbers of released components can thus be considered to be only informative annotations that help users in interpreting the status of a release. They have no distinguishing power, but nevertheless produce a lot of overhead when a release is brought out.

So we face a dilemma: either we release each component separately and release costs go up (due to the recursive nature of component-based releasing). Or we release all components at once, which is error-prone and tends to be carried out much less frequently.

Our aim in this paper is to explore a technical solution to arrive at feasible compromise. This means that we sacrifice the ability to maintain different versions of a component in parallel, for a more agile, less error-prone release process. The assumption of one relevant version, the current one, allows us to automate the release process by a continuous integration system. Every time a component changes it is integrated *and* released. From these releases we are then able to compute incremental updates.

2.2 Solution Overview

The basic architecture of our solution is depicted in Fig. 1. We assume the presence of a version control system (VCS). This system is polled for changes by the continuous release system. Every time there is a change, it builds and tests the components that are affected by the change. As such the continuous release process subsumes continuous integration [6]. In this paper, we mean by "integration" the process of building and testing a set of related components.

Every component revision that passes integration is released. Its version is simply its revision number in the version control system. The dependencies of a released component are also released revisions. The system explicitly keeps track of against which revisions of its declared dependencies it passed the integration. This knowledge is stored in a release knowledge base (RKB). Note that integrated component revisions could pass through one or more quality assurance stages before they are delivered to users. Such policies can easily be superimposed on the continuous release system described in this paper.

The RKB is queried by the update server to compute updates from releases. Such updates are incremental relative to a certain user configuration. The updates are then delivered to users over the internet.

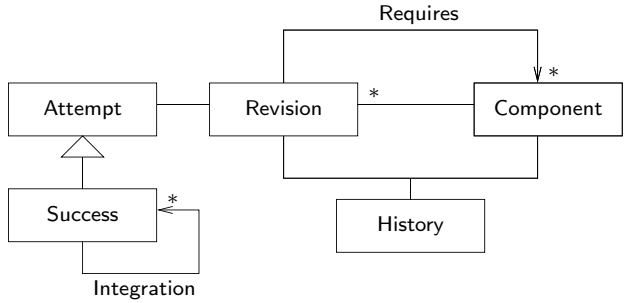


Fig. 2. Continuous Integration Component Model

3 Continuous Release

3.1 Component Model

Our formalisation is based on the calculus of binary relations [16]. This means that essential concepts are modelled as sets and relations between these sets. Reasoning is applied by evaluating standard set operations and relational operations.

We will now present the sets and relations that model the evolution and dependencies of a set of components. In the second part of this section we will present the continuous release algorithm that takes this versioned product model as input. As a reference, the complete model is displayed in a UML like notation in Fig. 2.

The most basic set is the set of components Component. It contains an element for each component that is developed by a certain organisation or team. Note that we abstract from the fact that this set is not stable over time; new components may be created and existing components may be retired.

To model the evolution of each component we define the set of component revisions as follows:

$$\mathsf{Revision} \subseteq \mathsf{Component} \times \mathbb{N}$$

This set contains tuples $\langle C, i \rangle$ where C is a component and i is a *revision identifier*. What such an identifier looks like depends on the Version Control System (VCS) that is used to store the sources of the components. For instance, in the case of CVS this will be a date identifying the moment in time that the last commit occurred on the module containing the component's sources. If Subversion is used, however, this identifier will be a plain integer identifying the revision of one whole source tree. To abstract from implementation details we will use natural numbers as revision identifiers. A tuple $\langle C, i \rangle$ is called a "(component) revision".

A revision records the state of a component. It identifies the sources of a component during a period of time. Since it is necessary to know when a certain component has changed, and we want to abstract from the specific form of

revision identifiers, we model the history of a component explicitly. This is done using the relation History, which records the revision a component has at a certain moment in time:

$$\mathsf{History} \subseteq \mathrm{Time} \times (\mathsf{Component} \times \mathsf{Revision})$$

This relation is used to determine the state of a set of components at a certain moment in time. By taking the image of this relation for a certain time, we get for each component in Component the revision it had at that time.

Components may have dependencies which may evolve because they are part of the component. We assume that the dependencies are specified in a designated file within the source tree of a component. As a consequence, whenever this file is changed (e.g., a dependency is added), then, by implication, the component as a whole changes.

The dependencies in the dependency file do not contain version information. If they would, then, every time a dependency component changes, the declaration of this dependency would have to be changed; this is not feasible in practice. Moreover, since the package file is part of the source tree of a component, such changes quickly ripple through the complete set of components, increasing the effort to keep versioned dependencies in sync.

The dependency relation that can be derived from the dependency files is a relation between component revisions and components:

$$\mathsf{Requires} \subseteq \mathsf{Revision} \times \mathsf{Component}$$

Requires has Revision as its domain, since dependencies are part of the evolution history of a component; they may change between revisions. For a single revision, however, the set of dependencies is always the same.

The final relation that is needed, is a relation between revisions, denoting the actual dependency graph at certain moment in time. It can be computed from Requires and History. It relates a moment in time and two revisions:

$$\mathsf{Depends} \subseteq \mathrm{Time} \times (\mathsf{Revision} \times \mathsf{Revision})$$

A tuple $\langle t, \langle A_i, B_j \rangle\rangle \in \mathsf{Depends}$ means that at point in time t, the dependency of A_i on B referred to B_j; that is: $\langle A_i, B\rangle \in \mathsf{Requires}$ and $\langle t, \langle B, B_j\rangle\rangle \in \mathsf{History}$.

3.2 Towards Continuous Release

A continuous integration system polls the version control system for recent commits and if something has changed, builds all components that are affected by it. After each integration, the system usually generates a website containing results and statistics. In this section we formalise and extend the concept of continuous integration to obtain a continuous release system.

The continuous release system operates by populating three relations. The first two are relations between a number identifying an integration attempt and

Algorithm 1 Continuous Integration

```
 1: procedure INTEGRATECONTINUOUSLY
 2:     i := 0
 3:     loop
 4:         deps := Depends[now]
 5:         changed := carrier(deps) \ range(Attempt)
 6:         if changed ≠ {} then
 7:             todo := deps^-1*[changed]
 8:             order := reverse(topsort(deps)) ∩ todo
 9:             INTEGRATEMANY(i, order, deps)
10:             i := i + 1
11:         end if
12:     end loop
13: end procedure
```

a component revision:

$$\mathsf{Attempt} \subseteq \mathbb{N} \times \mathsf{Revision}$$
$$\mathsf{Success} \subseteq \mathsf{Attempt}$$

Elements in Success indicate successful integrations of component revisions, whereas Attempt records attempts at integration that may have failed. Note that Success is included in Attempt.

The second relation records how a component was integrated:

$$\mathsf{Integration} \subseteq \mathsf{Success} \times \mathsf{Success}$$

Integration is a dependency relation between successful integrations. A tuple $\langle\langle i, r\rangle, \langle j, s\rangle\rangle$ means that revision r was successfully integrated in iteration i against s, which, at the time of i was a dependency of r. Revision s was successfully integrated in iteration $j \leq i$. The fact that $j \leq i$ conveys the intuition that a component can never be integrated against dependencies that have been integrated later. However, it is possible that a previous integration of a dependency can be reused. Consider the situation that there are two component revisions A and A' which both depend on B in iterations i and $i+1$. First A is integrated against the successful integration of B in iteration i. Then, in iteration $i+1$, we only have to integrate A' because B did not change in between i and $i+1$. This means that the integration of B in iteration i can be reused.

We will now present the algorithms to compute Success, Attempt and Integration. In these algorithms all capitalised variables are considered to be global; perhaps it is most intuitive to view them as part of a persistent database, the RKB.

Algorithm 1 displays the top-level continuous integration algorithm in pseudocode. Since continuous integration is assumed to run forever, the main part of the procedure is a single infinite loop.

The first part of the loop is concerned with determining what has changed. We first determine the dependency graph at the current moment in time. This

Algorithm 2 Integrate components

```
 1: procedure INTEGRATEMANY(i, order, deps)
 2:     for each r in order do
 3:         D := {⟨i, d⟩ ∈ Attempt | d ∈ deps[r], ¬∃⟨j, d⟩ ∈ Attempt : j > i}
 4:         if D ⊆ Success then
 5:             if INTEGRATEONE(r, D) = success then
 6:                 Success := Success ∪ {⟨i, r⟩}
 7:                 Integration := Integration ∪ ({⟨i, r⟩} × D)
 8:             end if
 9:         end if
10:         Attempt := Attempt ∪ {⟨i, r⟩}
11:     end for
12: end procedure
```

is done by taking the (right) image of relation Depends for the current moment of time (indicated by **now**). The variable *deps* represents the current dependency graph; it is a relation between component revisions. Then, to compute the set of changed components in *changed*, all component revisions occurring in the dependency graph for which integration previously has been attempted, are filtered out at line 5. Recall that Attempt is a relation between integers (integration identifiers) and revisions. Therefore, taking the range of Attempt gives us all revisions that have successfully or unsuccessfully been integrated before.

If no component has changed in between the previous iteration and the current one, all nodes in the current dependency graph (*deps*) will be in the range of Attempt. As a consequence *changed* will be empty, and nothing has to be done. If a change in some component did occur, we are left with all revisions for which integration never has been attempted before.

If the set *changed* is non-empty, we determine the set of component revisions that have to be (re)integrated at line 7. The set *changed* contains all revisions that have changed themselves, but all current revisions that depend on the revisions in *changed* should be integrated again as well. These so-called *co-dependencies* are computed by taking the image of *changed* on the transitive-reflexive closure of the inverse dependency graph. Inverting the dependency graph gives the co-dependency relation. Computing the transitive-reflexive closure of this relation and taking the image of *changed* gives all component revisions that (transitively) depend on a revision in *changed* including the revisions in *changed* themselves. The set *todo* thus contains all revisions that have to be rebuilt.

The order of integrating the component revisions in *todo* is determined by the topological sort of the dependency graph *deps*. For any directed acyclic graph the topological sort (topsort in the algorithm) gives a partial order on the nodes of the graph such that, if there is an edge $\langle x, y\rangle$, then x will come before y. Since dependencies should be integrated before the revisions that depends on them, the order produced by topsort is reversed.

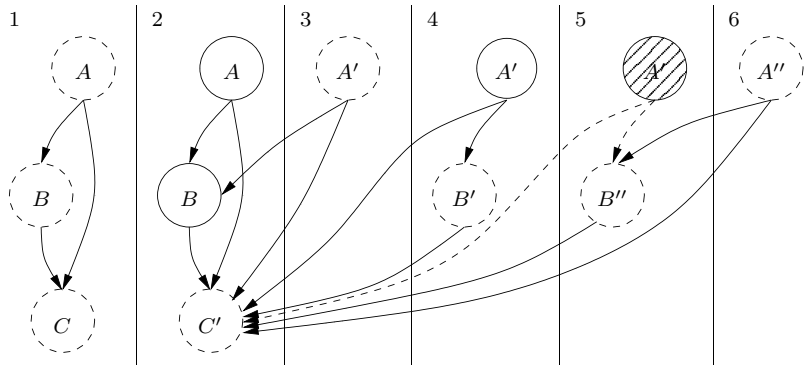


Fig. 3. Six iterations of integration

The topological order of the dependency graph contains all revisions participating in it. Since we only have to integrate the ones in *todo*, the order is (list) intersected with it. So, at line 8, the list *order* contains each revision in *todo* in the proper integration order.

Finally, at line 9, the function INTEGRATEMANY is invoked which performs the actual integration of each revision in *order*. After INTEGRATEMANY finishes, the iteration counter i is incremented.

The procedure INTEGRATEMANY, displayed as Alg. 2, receives the current iteration i, the ordered list of revisions to be integrated and the current dependency graph. The procedure loops over each consecutive revision r in *order*, and tries to integrate r with the most recently attempted integrations of the dependencies of r. These dependencies are computed from *deps* at line 3. There may be multiple integration attempts for these dependencies, so we take the ones with the highest i, that is: from the most recent iteration.

At line 4 the actual integration of a single revision starts, but only if the set D is contained in Success, since it is useless to start the integration if some of the dependencies failed to integrate. If there are successful integrations of all dependencies, the function INTEGRATEONE takes care for the actual integration (i.e. build, smoke, test etc.). We don't show the definition of INTEGRATEONE since it is specific to one's build setup (e.g. build tools, programming language, platform, searchpaths etc.). If the integration of r turns out to be successful, the relations Success and Integration are updated.

3.3 A Sample Run

To illustrate how the algorithm works, and what kind of information is recorded in Integration, let's consider an example. Assume there are three components, A, B, C. The dependencies are so that A depends on B and C, and B depends on C. Assume further that these dependencies do not evolve.

Figure 3 shows six iterations of INTEGRATECONTINUOUSLY, indicated by the vertical swimlanes. In the figure, a dashed circle means that a component has evolved in between swimlanes, and therefore needs to be integrated. Shaded circles and dashed arrows indicate that the integration of a revision has failed.

So, in the first iteration, the current revisions of A, B, and C have to be integrated, since there is no earlier integration. In the second iteration, however, component C has changed into C', and both A and B have remained the same. Since A and B depend on C', both have to be reintegrated.

The third iteration introduces a change in A. Since no component depends on A' at this point, only A' has to be reintegrated. In this case, the integrations of B and C in the previous iteration are reused.

Then, between the third and the fourth iteration B evolves into B'. Since A' depends on B', it should be reintegrated, but still the earlier integration of C' can be reused. In the next iteration B' evolves into B''. Again, A' should be reintegrated, but now it fails. The trigger of the failure is in B' or in the interaction of B' and C'. We cannot be sure that the bug that triggered the failure is in the changed component B''. It might be so, that a valid change in B'' might produce a bug in A' due to unexpected interaction with C'. Therefore, only complete integrations can be reused.

Finally, in the last iteration, it was found out that the bug was in A', due to an invalid assumption. This has been fixed, and now A'' successfully integrates with B'' and C'.

4 Continuous Upgrade

4.1 Release Packages

In this section we will describe how to derive incremental updates from the sets Success and Integration. Every element $\langle i, r\rangle \in \mathsf{Success}$ represents a *release* i of revision r. The set of revisions that go into an update derived from a release, the *release package*, is defined as:

$$\text{package}(s) = \text{range}(\mathsf{Integration}^*[s])$$

This function returns the bill of materials for a release $s \in \mathsf{Success}$.

As an example, consider Fig. 4. It shows the two release packages for component A'. They differ in the choice between revisions B and B'. Since a release package contains accurate revision information it is possible to compare a release package to an installed configuration and compute the difference between the current state (user configuration) and the desired state (a release package).

If ugrades are to be delivered automatically they have to satisfy a number of properties. We will discuss each property in turn and assert that the release packages derived from the RKB satisfy it.

Correctness Releases should contain software that is correct according to some criterion. In this paper we used integration testing as a criterion. It can be seen from the algorithm INTEGRATEMANY that only successfully integrated components are released.

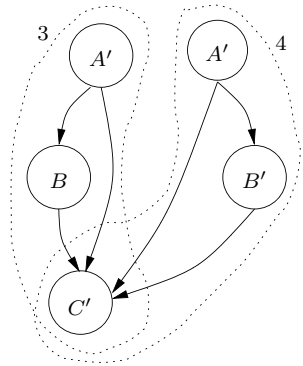


Fig. 4. Two release packages for A'

Completeness A component release should contain all updates of its dependencies if they are required according to the correctness criterion. In our component model, the source tree of each component contains a special file explicitly declaring the dependencies of that component. If a dependency is missed, the integration of the component will fail. Therefore, every release will reference *all* of its released dependencies in Integration.

Traceability It should be possible to relate a release to what is installed at the user's site in a precise way. It is for this reason that release version numbers are equated with revision numbers. Thus, every installed release can be traced back to the sources it was built from. Tracing release to source code enables the derivation of incremental updates.

Determinism Updating a component should be unambiguous; this means that they can be applied without user intervention. This implies that there cannot be two revisions of the same component in one release package. More formally, this can be stated as a knowledge base invariant. First, let:

$$\text{components}(s) = \text{domain}(\text{package}(s))$$

The invariant that should be maintained now reads:

$$\forall s \in \mathsf{Success} : |\text{package}(s)| = |\text{components}(s)|$$

We have empirically verified that our continuous release algorithm preserves this invariant. Proving this is left as future work.

4.2 Deriving Updates

The basic use case for updating a component is as follows. The sofware vendor advertises to its customers that a new release of a product is available [9]. Depending on certain considerations (e.g. added features, criticality, licensing etc.)

the customer can decide to update to this new release. This generally means downloading a package or a patch associated to the release and installing it.

In our setting, a release of a product is identified by a successful integration of a top component. There may be multiple releases for a single revision r due to the evolution of dependencies of r. The user can decide to obtain the new release based on the changes that a component (or one of its dependencies) has gone through. So, a release of an application component is best described by the changes in all its (transitive) dependencies.

To update a user installation one has to find a suitable release. If we start with the set of all releases (Success), we can apply a number of constraints to reduce this set to (eventually) a singleton that fits the requirements of a user.

For instance, assume the user has installed the release identified by the first iteration in Fig. 3. This entails that she has component revisions A, B, and C installed at her site.

The set of all releases is $\{1, 2, 3, 4, 5, 6\}$. The following kinds of constraints express policy decisions that guide the search for a suitable release.

- State constraints: newer or older than some date or version. In the example: "newer than A". This leaves us with: $\{3, 4, 5, 6\}$.
- Update constraints: never remove, or patch, or a add, a certain (set of) component(s). For example: "preserve the A component". The set reduces to: $\{3, 4, 6\}$.
- Trade-offs: conservative or progressive updates, minimizing bandwidth and maximizing up-to-dateness respectively. If the conservative update is chosen, release 3 will be used,—otherwise 6.

If release 3 is used, only the patch between C and C' has to be transferred and applied. On the other hand, if release 6 is chosen, patches from B to B'' and A to A'' have to be deployed as well.

5 Implementation

We have validated our formalisation of continuous release in the context the ASF+SDF Meta-Environment [17], developed within our group SEN1 at CWI. The Meta-Environment is a software system for the definition of programming languages and generic software transformations. It consists of around 25 components, implemented in C, Java and several domain specific languages. The validation was done by implementing a prototype tool called Sisyphus. It is implemented in Ruby[2] and consists of approximately 1000 source lines of code, including the SQL schema for the RKB.

In the first stage Sisyphus polls the CVS repository for changes. If the repository has changed since the last iteration, it computes the Depends relation based on the current state of the repository. This relation is stored in a SQLite[3] database.

[2] `www.ruby-lang.org`
[3] `www.sqlite.org`

The second stage consists of running the algorithm described in Sect. 3. Every component that needs integration is built and tested. Updates to the relations Attempt, Succes and Integration are stored in the database.

We let Sisyphus reproduce a part of the build history of a sub-component of the ASF+SDF Meta-Environment: a generic pretty-printer called `pandora`. This tool consists of eight components that are maintained in our group. The approximate size of `pandora` including its dependencies is ≈190 KLOC. The Sisyphus system integrated the components on a weekly basis over the period of one year (2004). From the database we were then able to generate a graphical depiction of all release packages. In the future we plan to deploy the Sisyphus system to build and release the complete ASF+SDF Meta-Environement.

A snapshot of the generated graph is depicted in Fig. 5. The graph is similar to Fig. 3, only it abstracts from version information. Shown are three integration iterations, 22, 23, and 24. In each column, the bottom component designates the minimum changeset inbetween iterations.

Iteration 22 shows a complete integration of all components, triggered by a change in the bottom component `aterm`. In iteration 23 we see that only `pt-support` and components that depend on it have been rebuilt, reusing the integration of `error-support`, `tide-support`, `toolbuslib` and `aterm`.

The third iteration (24) reuses some of these component integrations, namely: `tide-support`, `toolbuslib` and `aterm`. The integration of component `error-support` is not reused because it evolved in between iteration 23 and 24. Note that the integration of `pt-support` from iteration 23 cannot be reused here since it depends on the changed component `error-support`.

6 Related Work

6.1 Update Management

Our work clearly belongs to the area of update management. For an overview of existing tools and techniques we refer to [9]. Our approach differs from the techniques surveyed in that paper, mainly in the way how component releases and the updates derived from them are linked to a continuous integration process.

The package deployment system Nix [3] also automatically produces updates for components. This system uses cryptographic hashes on *all* inputs (including compilers, operating system, processor architecture etc.) to the build process to identify the state of a component. In fact this more aggressive than our approach, since we only use revision identifiers.

Another difference is that Nix is a generic deployment system similar to Debian's Advanced Package Tool [15], Redhat's RPM [1] and the Gentoo/BSD ports [14, 20] systems. This means that it works best if all software is deployed using it. Our approach does not prohibit that different deployment models peacefully coexist, although not across compositions.

Updates produced by Nix are always non-destructive. This means that an update will never break installed components by overwriting a dependency. A

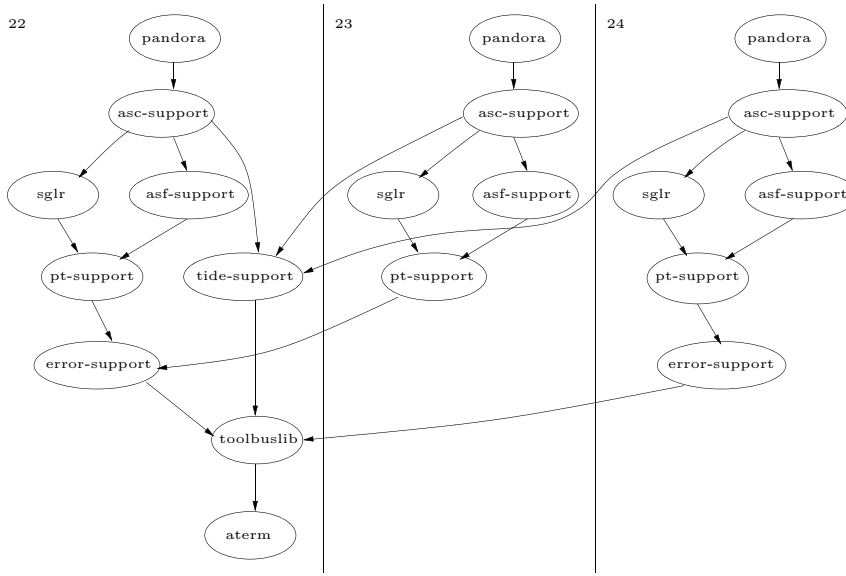


Fig. 5. Three weekly releases of the `pandora` pretty printing component in 2004

consequence of this is that the deployment model is more invasive. Our updates are always destructive, and therefore the reasoning needed to guarantee the preservation of certain properties of the user configuration is more complex. Nevertheless, this makes the deployment of updates simpler since no side-by-side installation of different versions of the same component is needed.

6.2 Relation Calculus

The relational calculus [16] has been used in the context of program understanding (e.g. [10, 12]), analysis of software architecture [8, 5], and configuration management [11, 2]. However, we think that use of the relational calculus for the formalisation of continuous integration and release is novel.

Our approach is closest to Bertrand Meyer's proposal to use the calculus for a software knowledge base (SKB). In [13] he proposes to store relations among programming artifacts (e.g., sources, functions) in an SKB to support the software process. Many of the relations he considers can be derived by analyzing software artifacts. Our approach differs in that respect that only a minimum of artifacts have to be analyzed: the dependencies between components that are specified somewhere. Another distinction is that our SKB is populated by a software program. Apart from the specification of dependencies, no intervention from development is needed.

7 Conclusion and Future Work

Proper update management can be a serious advantage of software vendors over their comptetitors. In this paper we have analysed how to successfully and quickly produce and deploy such updates, without incurring additional overhead for development or release managers.

We have analysed technical aspects of continuous integration in a setting of component-based development. This formalisation is the starting point for continously releasing components and deriving updates from it that are guaranteed to have passed integration testing.

Finally we have developed a prototype tool to validate the approach against the component repository of a medium-sized software system, the ASF+SDF Meta-Environment. It proved that the releases produced are correct with respect to the integration predicate.

As future work we will consider making our approach more expressive and flexible, by adding dimensions of complexity. First, the approach discussed in this paper assumes that all components are developed in-house. It would be interesting to be able to transparently deal with third-party components, especially in the context of open source software.

Another interesting direction concerns the notion of variability. Software components that expose variability can be configured in different ways according to different requirements [19]. The question is how this interacts with automatic component releases. The configuration space may be very large, and the integration process must take the binding variation points into account. Adding variation to our approach would, however, enable the delivery of updates for product families.

Finally, in many cases it is desirable that different users or departments use different kinds of releases. One could imagine discerning different levels of release, such as alpha, beta, testing, stable etc. Such stages could direct component revisions through an organisation, starting with development, and ending with actual users. We conjecture that our formalisation and method of formalisation are good starting points for more elaborate component life cycle management.

Acknowledgements Gratitude goes to Paul Klint, Jurgen Vinju and Gerco Ballintijn, who suggested important improvements to drafts of this paper. We thank the anonymous referees for providing many insightful comments.

References

1. E. C. Bailey. *Maximum RPM. Taking the Red Hat Package Manager to the Limit.* Red Hat, Inc., 2000. Online: `http://www.rpm.org/max-rpm` (August 2005).
2. E. Borison. A model of software manufacture. In *Proceedings of the IFIP International Workshop on Advanced Programming Environments*, pages 197–220, Trondheim, Norway, June 1987.

3. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In Lee Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
4. J. Estublier, J.-M. Favre, and P. Morat. Toward SCM / PDM integration? In *Proceedings of the Eighth International Symposium on System Configuration Management (SCM-8)*, 1998.
5. L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 4(28):371–400, April 1998.
6. M. Fowler and M. Foemmel. Continuous integration. Available at: `http://www.martinfowler.com/articles/continuousIntegration.html` (February 2005).
7. E. Grossman. An update on software updates. *ACM Queue*, March 2005.
8. R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, 1998.
9. S. Jansen, G. Ballintijn, and S. Brinkkemper. A process framework and typology for software product updaters. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, 2005.
10. P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.
11. D. A. Lamb. Relations in software manufacture. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, october 1994.
12. M. A. Linton. Implementing relational views of programs. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, Pittsburgh, PA, May 1984. Association for Computing Machinery, Association for Computing Machinery.
13. B. Meyer. The software knowledge base. In *Proc. of the 8th Intl. Conf. on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1985.
14. FreeBSD Ports. Online: `http://www.freebsd.org/ports` (August 2005).
15. G. Noronha Silva. *APT HOWTO*. Debian, 2004. Online: `http://www.debian.org/doc/manuals/apt-howto/index.en.html` (August 2005).
16. A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6:73–89, 1941.
17. M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
18. A. van der Hoek and A. L. Wolf. Software release management for component-based software. *Software—Practice and Experience*, 33(1):77–98, 2003.
19. T. van der Storm. Variability and component composition. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, June 2004.
20. S. Vermeulen, R. Marples, D. Robbins, C. Houser, and J. Alexandratos. *Working with Portage*. Gentoo. Online: `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=3` (August 2005).

Process Model and Awareness in SCM

Jacky Estublier and Sergio Garcia

LSR-IMAG, 220 rue de la Chimie
BP53
38041 Grenoble Cedex 9, France

{Jacky.Estublier, Sergio.Garcia}@imag.fr

Abstract. The development of large and complex systems, under hard time constraints, requires the participation of many developers working concurrently. SCM systems allow concurrent access to software artifacts, but provide poor support to maintain data consistency when concurrent changes are performed on the same artifacts. This problem can be reduced if developers are aware of the others work and warned about the conflicts that may arise, allowing the users to manage the risks more effectively.
Awareness, without any knowledge about the cooperative process and system models cannot help much, and indeed is not very much used today. We claim that awareness takes its potential only when it takes into account the cooperative process, and the system model in use. This paper, based on the experience gained with our tool Celine, explores the relationships between awareness, process and system models, and shows how the knowledge of these models can be used to improve the relevance of an awareness system.

1. Introduction

One of the key factors for the success of software in a competitive environment is time to market. This fact, added to the increasing size of software systems, leave organizations little choice other than to increase the number of engineers that work on a same product at the same time [5]. Unfortunately, rising concurrency increases the possibility of inconsistent modifications and puts the software stability at risk. SCM systems allow concurrency, but provide simplistic and inadequate support to handle this risk.

The databases community has in depth studied the problem of data concurrent modifications. There are, however, fundamental differences that make concurrent software engineering a field of research on its own. Those differences can be summarized in the following two points [12]:

1. Consistency definition. In software engineering there exists no adequate global criteria, such as serializability, to enforce the correctness of concurrent modifications. Serializability is not adequate because software is composed of complex and tightly interrelated data, such that most modifications have potential side effect on a large and a priori unknown sub set of the data. This means that two concurrent

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

modifications are almost never serializable under the classic databases definition [12][13].

2. Long duration. In databases, it is common to abort a failed transaction. Engineering tasks take days of human work, which makes it unacceptable to drop them. [12]

Usual SCM tools allow concurrent work by providing private workspaces and mergers to try to reconcile concurrent work; but they do not provide effective mechanisms to coordinate developers working concurrently. Without such mechanisms, project managers must either impose simplistic restrictions on concurrency (file locking) or they must leave all the burden of avoiding data corruption to the developers. When the number of developers increases, the risk of modifications being combined inconsistently increases.

It has been argued that this problem can be palliated by raising the awareness of developers about the work of their partners [1][3]. The main hypothesis is that if developers are given a continuous insight over the group activities, they can detect and handle potential conflicts more effectively during the development process. The challenge of an awareness system is to provide the user with the relevant, and only the relevant information. This paper explains how process and data models are related to awareness and how their knowledge allows building a smart awareness system.

Section 2 presents some background and section 3 explains what is the purpose of awareness. Section 4 relates cooperative process with awareness, 5 with cooperative policies and business processes. Section 6 introduces shortly the relationship between system model and awareness. Section 7 presents Celine an section 8 concludes.

2. Background

2.1. Cooperative engineering

We can characterize cooperation as a group of persons pursuing the same goal. In software engineering, the goal is to transform a software application from its actual state say V0 into a new state say V1.

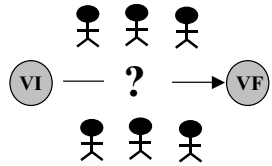


Fig. 1 Cooperation

The role of cooperative software engineering is to control and support how this transformation happens [8]. There exist several basic strategies. For instance, work can be serialized, allowing only one developer to modify the software at a time.

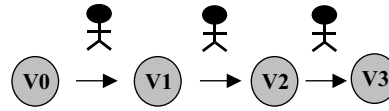


Fig. 2 Serialization

To increase the development speed, the work load can be broken down in several tasks and subtasks that are assigned to different developers and executed in parallel. Those tasks should be designed such that they involve different parts of the system; unfortunately, provided the strong coupling between the software artifacts, it is likely that more than one developer will need to change the same software artifact. Concurrent change of the same artifact must be possible. But since most development activities require exclusive access to the data, concurrent software engineering needs a system that supports multiple copies, called cooperative copies, of the developed software. Concurrent engineering is about the control of how these cooperative copies are created and reconciled into a common result.

2.2. Asynchronous Cooperation with Workspaces

SCM is not the only field supporting cooperative changes; indeed all engineering domains face the cooperative engineering challenge. It is the nature and relationship between data, the nature and duration of activities and the development process that makes that some solution are relevant in some engineering domains but not in others. For example, Computer Supported Cooperative Work (CSCW) systems aim at controlling how a document is co-authored, which looks pretty close to software engineering, but CSCW often follows a blackboard metaphor where multiple participants work simultaneously on the same document. In those systems, even the smallest modification is visible to anyone "as soon as" it occurs. Cooperative editors implement that strategy propagating “instantaneously” all changes over the network, for all the document views to be permanently synchronized. [16] [17]

The CSCW approach is not adequate for software engineering because most software changes are performed as a set of partial modifications that make sense only within the context of a single task. Interleaving these partial modifications would therefore produce a document in an inconsistent state, prohibiting compilation and test until the end of the whole work. In software engineering, instead, each task makes sense by itself (e.g. fixing a bug), and should be compiled and tested independently. Such tasks require time and, usually, the availability of a significant fraction, if not all, the code.

This is precisely the role of workspaces: to host complete, isolated copies of the software during arbitrarily long periods of time, allowing several independent development activities to be performed simultaneously and independently. Developers can modify their private copies, leave them in inconsistent states for arbitrarily long periods of time, perform operations on them without being subjected to the others changes, and vice versa [1][4]. This separation between private local copies and public copies is known as workspace isolation.

Unlike CSCW systems, in software engineering, the synchronization among the different workspaces is done explicitly and usually involves a complete and consistent unit of work. What means “complete and consistent” is highly related to the workspace purpose and development process. A large part of cooperative engineering *policies* consists in defining “complete and consistent”, and to define between which workspaces synchronization should occur.

2.3. Divergence and reconciliation

Our hypothesis of a final common result implies that eventually all the cooperative copies are merged into the final result. Therefore cooperative engineering is fundamentally about merge control. The problem is twofold:

- **Data consistency** : Does merge provides a consistent result?
- **Process** : Are changes performed by the right person, on the right artifact at the right time?

The two points seems unrelated but experience shows that the process purpose is to make in such a way merges have the best chance to produce consistent result, and to be performed by the persons capable to solve conflicts if any. [2]

Serialization being the only well known consistency criteria, the merge function is a substitute for serialization : it takes two changes performed independently and concurrently on two copies of the same data and is supposed to produce the same result as if the changes have been produced in sequence i.e. the “second” change being performed on the data produced by “first” change. Unfortunately, the merge function depends on the nature of the data. For example for sets, the merge function exists and always produces the right result; for lists, only approximate merge functions exist; for complex data structures, only specialized algorithm taking into account the data semantics can perform merges. Programs are often assimilated as lists (of lines code) for which only an approximate algorithm exists; language specific mergers also exist but do not provide much better result and therefore are not used in practice [14].

Line merging is therefore a difficult and error prone task that needs human expertise and understanding of both the software and the modifications. When two modifications are found incompatible, large parts of them might have to be discarded, with the loss of hours or days of work. The risk of incompatible changes, and the complexity of merge, increases with the amount of changes to be merged. It is well known that, if waiting too much before to merge, the merge becomes very complex and risky, if not impossible. The only solution is keep divergences “small enough” by merging the different copies as soon as possible. These early merges create intermediate "agreement" points that serve as the base for the next reconciliation, reducing the gap that needs to be closed each time.

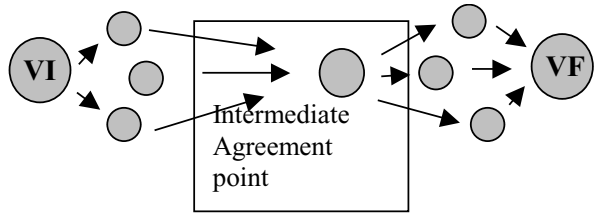


Fig. 3 Cyclic reconciliation.

As a consequence, cooperative engineering is a compromise between (1) maintaining isolation until the complete work is done and (2) merging as often as possible. It is the development process that defines and controls that compromise.

2.4. The development processes

How data is modified, when, by who, and how work is combined towards a single result is what we call the development process. From a data flow perspective, the development process can be represented by a directed graph, where nodes represent workspaces, and arcs represent the data flowing from origin node to destination node.

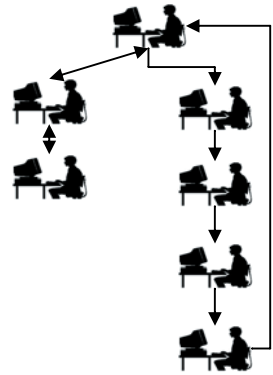


Fig. 4 Example of a static graph of workspaces

The development process can be controlled by different processes defined at different levels :

1. *Concurrent Engineering process*, which defines the graph topology i.e. its nodes and arcs. This graph defines how team work is structured and which paths the data can follow.

2. *Policy process,* which manages the data flow along the cooperative graph.

3. *Business process* which includes the change control process, engineering methodology, business activities, i.e. which activities are to be performed on which entities, for which reason, by whom and so on.

In this paper we will focus in the first level only, but our tool Celine covers point 1 and 2 [6], and is integrated with Apel [7] and Melusine [9], which cover point 3.

3. Awareness : purpose and issues

Given that the merge algorithm for software source code is not perfect, merges can be made safer enforcing the following practices:

- Merge often, so conflicts are easier to solve.
- Allow only concurrency on unrelated changes, so to reduce the probability of conflict.
- Give the task to merge to the person most likely to perform the right job.

A mixture of these strategies can be provided either by implementing a well defined process (the concurrent engineering policy) or by trusting the developers to apply the safe practices by themselves. Unfortunately, without any information about the work of their partners; developers do not know what are the current changes, who did them, for what purpose and so on. Therefore developers cannot apply the above practice. This is why many companies simply prohibit concurrent engineering for being too risky. The purpose of awareness is to give the developers the necessary context and information allowing them to apply the above good practice, making concurrent engineering "safe enough" for the advantages of a faster development significantly outweigh the risk of inconsistent or impossible merges [3].

Since the problem of safe concurrent engineering is clearly the problem of controlling the merge, the main criteria to evaluate the effectiveness of an awareness system is to measure to what extend the system helps the developer in deciding what to modify and when to merge.

We call *absence of concurrent engineering process* the fact modifications can flow without control between any pair of workspaces. Under this chaotic model, data can follow any possible path, making it impossible to foresee how and where a merge will occur. Without process, an awareness system can only inform all the users about the differences among all the existing work copies.

For a large numbers of workspaces, not only the system cannot give any hint on what is likely to occur (anything can occur), but also displaying this raw information leads to the problem of cognitive overload [1]: when too much information is presented, the cost of understanding it is too high for awareness to be of any value. For example, suppose 50 concurrent workspaces containing 100 files each, and for an average of 1 changes per file[1], a classic awareness system will display 5000 awareness messages which is much too much to consider; further, if you do not know

[1] These values are those of the application on which Celine is used today and are typical of many software development projects; but our Dassault Systems customer has 1000 concurrent workspaces, containing 2000 files in average, with 3 changes in average for each file, i.e. 6 millions awareness messages to display to each developer!.

why these changes are performed, and in which state they are, the information is of little help.

So far, all the awareness systems we know are ignoring the process, and therefore are of little value. We believe it is the reason why these systems have not been adopted by practitioners. An awareness system, to succeed, must provide simultaneously much less information and much more relevant information taking into account the process and the product models, in order to forecast the upcoming merges. The relationship between awareness and process models, and between awareness and system models will be presented in the following sections.

4. Concurrent Engineering Process and Awareness

4.1. Concurrent Engineering Process

The *concurrent engineering process* defines how workspaces are organized to communicate their work. This process defines which workspace can send its artifacts to which other workspace.

Concurrent engineering is usually not chaotic, but relies on a graph that guarantees some properties. For example, CVS, RCS, Oracle 8 workspaces, as well as most SCM systems are based on a star topology: a central database serves as a hub for multiple "satellite" workspaces that are not allowed to communicate directly. This approach is attractive because it ensures that the central database will always contain a copy that is not too old compared to the developer's copies, making it optimal to represent the collective work of the team at any time.

Unfortunately, for highly collaborative projects the number of workspaces under the central database can quickly become too big to keep awareness information tractable.

A concurrent engineering process well suited for awareness must satisfy the following properties:

1) Scalability: the process should be able to host a large number of workspaces while keeping the amount awareness information tractable.
2) Merge control relevant information: the process should provide the means to forecast when, where and whom will perform merges.

4.2. Scalable processes : Groups.

We have already mentioned that a cooperative work aims at producing a single result called the project's *reference copy*. Conversely, we call *work copy,* the copy that each developer can directly modify and that contains his/her work.

In our work, we made the hypothesis that the reference copy is hosted by a well identified workspace, called the reference workspace. The reference workspace

contains at any time, the actual visible result of the collective work. This hypothesis is often satisfied in practice, because software development is a continuous process where successive reference copies need to be produced, each developer feels more comfortable if is known where is the actual reference copy. In most usual systems, the data base plays the role of reference workspace.

We call a *group* a set of workspaces that have an explicit reference workspace, and where only the reference workspace communicates with the other workspaces.

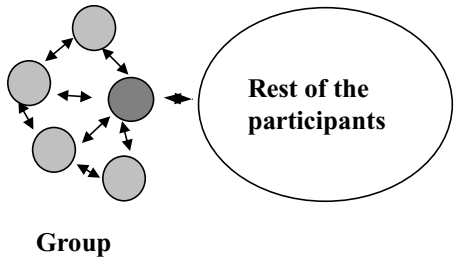


Fig. 5 Group

Groups are useful in practice because [12]:

- Groups easily match work division (it can be in charge to realize a task).
- Groups may use different concurrent engineering policies.
- Groups facilitate overall management, by hiding their operational details from the rest of the organization.

Groups easily match administrative and geographical divisions within an organization.

Interestingly from the awareness perspective, a group can also be organized hierarchically by allowing a workspace in the group to become the reference workspace of a sub-group. Such a hierarchy permits partitioning a task into sub-tasks. At Dassault Systems, the development of the Catia system uses a 7 level group hierarchy, allowing different types of concurrent engineering policies ranging from very strict at the top levels where are located the “Best So Far” and “Public” releases, to light constraints “in the trenches” : the development workspace groups at the bottom of the hierarchy [6][18].

From awareness perspective, a groups hierarchy based on tasks and subtasks is interesting because it makes groups a good candidates as the context for users to be aware of. Limiting awareness to a group activities raises the scalability of the system: the amount of information depends only on the number of workspaces within a group not on the total number of workspaces.

Modifications in the different workspaces of a group are usually semantically related because groups are created to perform particular tasks cooperatively, and therefore the likelihood to work on overlapping sets of files is pretty high. A group

member can examine his partner's modifications under the light of the group's mission, which provides an help in understanding its logic and intention.

4.3. Distance and States

Groups give scalability to awareness, but without a mechanism to make group awareness more precise about the forthcoming merges, the awareness information remains too broad and the developer has difficulties to measure to what extend others changes have consequences in his/her personal work.

We need a mechanism that indicates to the developer where and how divergences between his and somebody else work will be merged. Group awareness can be made richer by using the properties of the internal process.

We call *distance* between a source copy and a destination copy, the number of nodes (workspaces), along the cooperative graph, the first copy has to cross before to meet the destination copy.

Each time a copy moves from a node to the next one along the graph, the changes performed in the source node are potentially discarded and/or merged with the work performed in the new node. Therefore, for a long distance, the changes performed in the source copy are likely to reach the destination node much later and significantly transformed. The analysis of "far away" changes is likely to be at least partially irrelevant.

The shorter the distance, the more urgent and relevant the divergence analysis.

The distance is a good measure of the awareness information relevance; it is the main way an awareness system takes vantage of the cooperative process information.

As mentioned above, the cooperative policy defines when an through which node a change will move from its actual location to the next one. It is the combination cooperative graph, cooperative policy that defines the distance. The distance being determined by the cooperative graph and cooperative policy, the selection of a cooperative graph is of critical importance for concurrent engineering and awareness.

In the absence of cooperative policy (which is the usual case), with only the cooperative graph as information, the minimal distance is easy to compute, but the maximum distance in the general case is infinite, and the real distance unpredictable.

For example, on one extreme, for a graph being a line or a circle, the awareness system can computes distances, i.e. it can exactly forecast when a given change will reach a given developer. In that case, the information is only for information purpose since developers have no way to influence the distance. On the other extreme, without graph (each node is liked to all the others), and without policy, the minimum distance is always 1, but the real distance is totally unpredictable. This context makes the developed somehow nervous, since each change can be of maximum relevance (the distance can be 1) but since the real distance in unknown, the relevance is unknown; they are too many information to consider and no hint to know which one to consider first. The awareness information is useless.

4.3.1. The simple star topology

For the star topology, where the reference workspace is in the center, the minimum distance is one with the reference workspace, and two for the other workspaces. This simple topology is very much used, since it clearly identifies a unique workspace (the reference workspace) as being more relevant than any other one. If each group uses a star topology, the whole workspace structure is a tree of groups.

We define the concept of state of an artifact copy as

- The distance between that copy change and the reference copy,
- The distance between somebody else change and that copy.

In a star topology, from the point of view of a workspace different from the reference workspace, a change can only be in one of the following state :

- Distance to the reference : 0 (Unchanged) or 1 (Modified)
- Distance from a foreign change : 1 from the reference workspace (Obsolete), 2 from any other workspace (Changed).

In Celine, we call these states respectively modified, obsolete and changed, for user convenience, the combination (Modified / Obsolete) is called *Conflict* which means next move of that copy to or from the reference workspace will require to perform a merge.

For example, a user that is aware of a “conflict” state might decide to perform an early merge. A “changed” file might require no particular attention from the user, but a dangerous combination of “changed” and “modified” states might trigger the decision of one of the developers to stop further modifications of a file, or to both developers to exchange information to solve the emerging conflict early.

4.4. The extended star topology (Public and private workspaces)

The star topology is one of the most popular because it separate changes in two groups : those promoted to the reference workspace are implicitly made “official” and available to the rest of the group members, and those still in the developer workspace are supposed to be underway changes, with unknown level of stability and permanence. This difference is well expressed by the distance 1 with the reference, and 2 with the underway changes.

It would be interesting to have more complex types of processes within a group allowing more variability in distance to classify divergences as more or less urgent/relevant. The extended star topology is a good example of such a process.

In an extended star, each workspace is made of two parts : the first one (private) contains the current state of the developer’s work in progress, and the second one (public) holds the latest snapshot that the developer wants to share with the rest of the group. With respect to the simple star, this topology has the following advantages :

- The developer can make public a “stable” version of his/her work and continue working.

- Making public a version can be performed even when working off line.
- The promotion of the public versions into the reference workspace can be done at a later time, in an order defined by the policy or by an "integrator" working in the reference workspace.

In the reference workspace, the same occurs, the promotion occurs in the private area, and can be made public only after some work (e.g. validation) have been performed. In this topology the distances and states, are as follows :

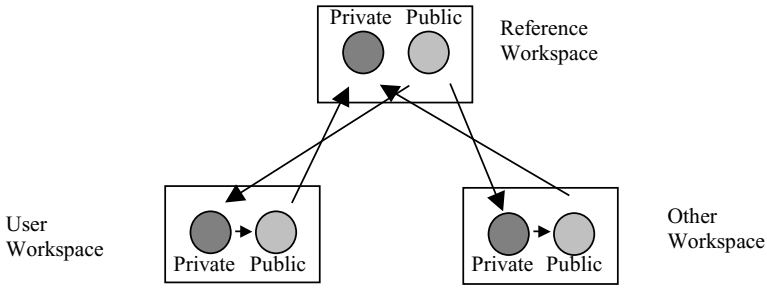


Fig. 6 Extended star graph.

- To reference workspace : 1 from public (Modified), 2 from private (Locally Modified)

From others workspace : 1 from public reference workspace (Obsolete), 2 from private reference workspace (Locally Obsolete), 3 from developer public workspace (Changed), 4 from developer private workspace (Loc*ally Changed*).

Clearly, the relevance of someone else change is well represented by the distance (four levels), but the number of states is now larger, especially if combined with the state of its own changes (three levels including the unchanged one) (4*3 = 12) . Even with this simple topology, the number of states increases pretty fast, making complex graph without cooperative policies intractable, falling back again in the cognitive overload mentioned above.

5. Policies, business processes and awareness

5.1. Cooperative Engineering Policies

A concurrent engineering policy is a process that enforces a particular strategy for concurrent engineering, by the definition of the path that can be followed in the cooperative graph, between two nodes. From a practical point of view, cooperative policies are implemented using a "Lock" primitive, which reserve the right to change a copy to a single workspace in a group, and controlling which operation (like promote, synchronize, publish, etc) are allowed given the circumstances. [6][11]

Whatever the implementation, a policy constrains the distance to a narrow range of values, if not a single one, even in complex graph. A policy is a way to use flexible and complex graph, still maintaining the distance into predictable are relatively small values.

If the awareness system knows the distance, as constrained by the cooperative policy, the number of states to manage can be small enough for proposing a good awareness system simple, relevant and useful. Indeed, In the absence of policy, only very simple graphs, like those presented above can be used, and awareness is a substitute to explicit policies : the visibility provided by awareness can be used by developers to decide themselves what is the policy to use at any point in time.

Except Celine, we do not know any system that propose explicit and high level cooperative policies, and that couples policy and awareness. This topic of defining and implementing policy in Celine was the topic of a previous paper [6], but our way to define policies will be modified, to improve and generalize the way policy and awareness can work in symbiosis.

5.2. Awareness and business models

Policy models are process models intended to control the data flow along the cooperative graph. Their main purpose is to make cooperative engineering safer, through a better control of merges, and incidentally to improve awareness functionalities.

Change control is the process that defines which change to implement, which groups to create to support these changes, with which policy and so on. It is our claim that change control would highly benefit from the existence of explicit cooperative policies, cooperative processes, and awareness, as presented in this paper. Our system Apel, since long, have addressed these issues [7].

Business process cover more high level processes addressing other topics, not necessarily related to changes. Our Apel system have been designed with these issues in mind, but is not discussed here.

It is our claim that these different levels of processes cannot be addressed in a single formalism, but benefit from each other, and should be connected in some way. Our work on Mélusine address this issues of heterogeneous processes interoperability [9].

6. System models

A system model is the definition of the relationships between the entities forming a system. In software engineering it includes the definition of the relationships between the different files present in a workspace. Some relationships may indicate a strong semantic dependency between files, meaning that a change in one of them, semantically, is also a change in the other one, called indirect change.

This information is of importance for awareness for two fundamental reasons :

- Changes are not only direct but also indirect changes,
- Granularity of operations like promote, synchronize or lock is not the file, not the whole workspace, but the transitive closure of the dependency operation.

System model information is very important for cooperative engineering control, since it increases the relevance of the information provided. Without system models, systems are taking conservative policies : the granularity of operations (promote, ..) is the whole workspace and only direct changes are considered by the awareness system.

We have defined an extension of the Eclipse framework for the complete support of system models and we are in the process of integrating Celine with this workspace system. We hope this experience will be reported in a future paper.

7. Celine

The Celine system has been built initially for the support of cooperative engineering policies [6][10]; once deployed we have realized that most companies are not yet ripe for policies, and prefer to rely on the developers expertise to handle cooperation, provided that the right information is available to them. This is why we have added awareness to Celine. We have first hard wired the star topology, then the extended star one, and integrated with cooperative policies.

Celine is daily used in production at STMicroelectronics by a group of engineers that with 70 workspaces (in average) collaborate on the development of large number (a few hundred) of microelectronic design files that for all purposes can be considered as software source code. Within those files, a kernel group of around 100 files is under daily modification. The number of concurrent modifications is high for the kernel files (50 concurrent changes in average), and the number of developers in the team make impossible to rely on informal communication channels to find out who is modifying what. Indeed, before to use Celine, that group was forced to lock file (using the synchronicity version control system), and therefore to avoid concurrent engineering, with high penalty in overall productivity.

In that team, the topology of the development process is a simple star. Celine provides each user with a view of the data that exposes the different state of each file, as explained in section 4.3.

In average, an engineer has 5 to 10 active simultaneous workspaces. To reduce the amount of awareness information, we encapsulate the workspaces of a particular user as a single source of divergences, displaying a single icon for a file even when present in more than one workspace. The Celine interface provides a way to find precisely who are the participants that have performed modifications and also to visualize the differences between any of them and the user's local copy.

Celine natively supports the star and extended star topology, on top of a CVS or synchronicity repositories. Celine handles and supports the basic operations (promote, synchronize, lock and so on), as part of its cooperative engineering policy (shown in the right menu bar in the above picture).

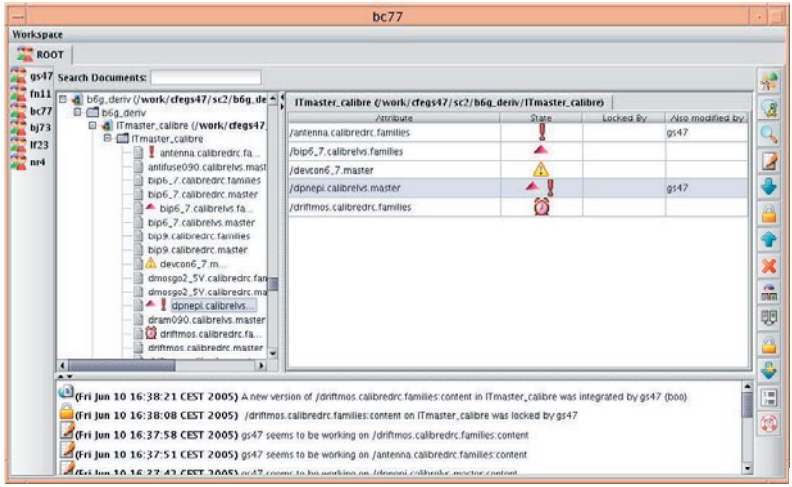


Fig. 7 Celine

Since arbitrary cooperative graphs are special cases of cooperative policies, we envision to extend Celine by a deeper integration with the cooperative policy support layer. The goal is to deduce automatically from the cooperative policy what are the possible states, and to generate a specific awareness support optimized for that policy.

Currently, Celine is in the process of being integrated with workspace supporting system model (as an extension of Eclipse). The current evolution of Celine is to support uniformly all the levels of process, individually or in combination, and to take vantage of the system model to generate an optimal and highly relevant awareness information.

8. Conclusion

Large software development projects need some concurrent engineering control to reduce the risks inherent to simultaneous modification of sensitive artifacts (e.g. source code). It is clear than a pessimistic (locking) strategy is not only too restrictive but also inadequate as it does not take into account semantic relationships among the different software elements.

In the absence of adequate means to control concurrent engineering, all the burden of dealing with copy reconciliation is on the shoulders of developers. Worse, conflicts are detected only when copies are merged, not when conflicts are created. Awareness has been proposed as a mechanism supporting concurrent engineering, providing developers with continuous insight of the team's activity. The hypothesis is that developers, with that information, will anticipate conflicts, and will reduce the probability of tricky or impossible merge, making cooperative engineering “safe enough”.

The above hypothesis does not hold in the general case, because it generates too much and not relevant enough information. As a matter of facts, these systems exist but are not used. We have shown that cooperative engineering relies on different models:

- Cooperative graph model (workspace topology)
- Cooperative policy model (allowed path along the graph)
- Business process model (partial order of activities)
- System model (data dependency)

Our claim is that, to be relevant and useful, an awareness system must take into account information coming from these models.

The cooperative graph introduce the concepts of group and distance. The group concept allows scalable system and solves the awareness cognitive overload issue. We have shown that an advanced awareness system can automatically transform the distance into a state displayed to the developer. We have shown why distance is the relevant information from cooperative engineering point of view, but distance can be computed only for simple graphs.

Cooperative policies complement the cooperative graph, allowing to control complex and flexible graphs, still maintaining short and predictability distances. Policies allow to provide a relevant awareness information even using complex cooperative graphs. Policies and awareness are two ways to provide cooperative engineering control; policies providing and explicit and imperative process, awareness relying instead on the developer expertise. Policy and awareness seem opposite, but our experience shows they are complementary, the policy providing the way to provide relevant awareness information, even in complex situations, which in turn allows to let large parts of the process under the developer responsibility.

The Celine system is, to our knowledge, the first system that provides awareness taking into account all the above models, and which is in daily industrial production. The experience so far shows that awareness along with the other models provide a very good solution to concurrent engineering; while each one of these model, alone, falls short to address the complexity and variability of the concurrent engineering challenge.

References

[1] Sarma, A., Noroozi Z., Van Der hoek, A.: "Palantir: Raising Awareness among Configuration Management Workspaces". 25th International Conference on Software Engineering. 05 03 – 05, 2003. Portland, Oregon

[2] Cleidson R.B. De Souza, David Redmiles, Gloria Mark, John Penix, Maarten Sierhuis. “Management of interdependencies in Collaborative Software Development”. 2003 International Symposium on Empirical Software Engineering (ISESE'03)

[3] Paul Dourish, Victoria Bellotti. "Awareness and Coordination in shared workspaces" Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)

[4] Cleidson R. B. De Souza. David RedMiles, Paul Dourish. “Breaking the Code, moving between private and public work in Collaborative Software Development" ACM SIGGROUP Bulletin Volume 24 , Issue 1 (April 2003)

[5] Dewayne E. Perry, Harvey P. Siy, Lawrence G. Votta. "Parallel Changes in Large Scale Software Development: An observational Case Study" ACM Transactions on Software Engineering and Methodology (TOSEM)

[6] Jacky Estublier, Sergio Garcia, German Vega. "Defining and Supporting Concurrent Engineering policies in SCM" SCM-11 May 2003, Portland, Oregon, USA

[7] J. Estublier, S. Dami and M. Amiour "APEL, an effective Process Support Environment." First International workshop on Multi Facets Software Process Engineering. September 22-23. Tunis, Pages 123, 137. Tunisia. 1997.

[8] Bischofberger W.R. Kleinferchner C.F. Mätzel K.-U. "Evolving a Programming Environment Into a Cooperative Software Engineering Environment" Proceedings of CONSEG '95, New Dehli, February 1995, pages 95-106, Tata McGraw-Hill, 1995.

[9] J. Estublier, G. Vega "Reuse and Variability on Large Software Applications” ESEC/FSE, September 2005, Lisboa Portugal.

[10] J. Estublier “Objects control for software configuration management”. 8 June 2001. CaiSE 2001, Interlaken, Suisse.

[11] A. van der Hoek, A. Carzaniga, D. Heimbigner, A.L. Wolf. "A Testbed for Configuration Management Policy Programming," IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 79-99, January 2002.

[12] M.H. Nodine, S. Ramoswamy, and S. Zdonik. A Cooperative Transaction Model for Design Databases. In Database Transaction Models, pages 59--86. Morgan Kaufmann, 1992.

[13] A. Skarra. "Localized correctness specifications for cooperating transactions in an object-oriented database". Office Knowledge Engineering, 4(1):79-106, 1991.

[14] T Mens “A state-of-the-art survey on software merging” - IEEE Transactions on Software Engineering, 2002. 28(5): p. 449-462

[15] P. Molli, H. Skaf-Molly and G. Oster. “Divergence awareness for virtual team through the web”. Proceedings of the integrated design and Process technology, 2002.

[16] C. Sun and C. A. Ellis: "Operational transformation in real-time group editors: issues, algorithms, and achievements, " In Proc. of ACM Conference on ComputerSupported Cooperative Work, pp.59-68, Seattle, USA, Nov. 1998.

[17] M. Beaudouin-Lafon and A. Karsenty. “Transparency and awareness in a real-time groupware system”. In 5th annual ACM symposium on User Interface Software and Technology. ACM Press 1992

[18] J. Estublier “Distributed Objects for Concurrent Engineering”. International Symposium, SCM-9, Toulouse, France, September 1999. Proceedings

Towards a Suite of Software Configuration Management Metrics

Lars Bendix, Lorenzo Borracci

Department of Computer Science, Lund Institute of Technology,
Box 118, S-210 00 Lund, Sweden
bendix@cs.lth.se, loreborra@gmail.com

Abstract. Software Configuration Management (SCM) is an important support activity in software development. However, its transparent nature as a service that makes life easier for others and as an insurance against disasters, often makes it difficult to justify investments in tools and processes that apparently do not have any direct return. We have made a first step towards establishing a model for showing the return on investment in SCM, making the costs and benefits explicit. In this paper, we also sketch how we plan to take the next important step and establish a set of metrics that can be used to manage and tune the SCM processes and tools.

1 Introduction

Development and maintenance of software require the use and interaction of many disciplines and groups of people. In the CMM model Software Configuration Management (SCM) is one of the key process areas that needs to be mastered to progress from the initial level of ad-hoc chaos to level 2 (Repeatable). However, unlike other key process areas, such as requirement engineering, SCM is not a discipline that ends up with a tangible product that can be weighed and judged for cost and quality. It is transparent (service and insurance) yet vital to help other disciplines carry out their work. Most of its benefits are "invisible" and spread over many groups (budgets), whereas the costs are mostly explicit and placed in the SCM group. So how do we justify initial or further investment in SCM?

It would be nice if there existed a simple model to calculate the Return On Investment (ROI) for SCM. Of previous academic work we are only aware of Larsen and Roald [LR98], that does not give a complete model, but measures some data pre and post implementation of an SCM tool and process. Many tool vendors have some promotional material, that use a simple ROI model [Merant01], but these models are not complete, nor are the values they claim credible. Such simple models may be adequate to justify the initial introduction of SCM, but they are not sufficiently complete to calculate the ROI of upgrading the SCM tool or introducing new SCM processes.

We want to do a more thorough analysis of the costs and benefits of investing in SCM to get a more complete picture. We are well aware that it is very difficult to get the full picture and that many costs and especially benefits are subjective and/or hard to quantify. This has to be dealt with in some way in such a model. Such a ROI model

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

can have many uses and must be tailored for that. In our model, we aim both at companies that want to introduce SCM and at companies that want to upgrade their SCM tool and/or processes. Our motivation for doing this is the simplicity of handling only one model and the fact that it is indeed possible to use a complete model to establish the costs and benefits and calculate the ROI both for initial introduction of SCM and for later adding to a partial implementation of SCM.

However, the ROI model is not our ultimate goal. We find it more interesting to go one step further and provide help for the SCM manager to daily monitor, manage and tune their SCM tool and processes – and to be able to predict specific SCM costs and benefits for projects. Therefore we are looking for a set of SCM specific metrics. Traditionally metrics and SCM is thought of as data supplied by SCM to monitor other software development processes and not the SCM processes. In this work in progress, we will treat the ROI model as an intermediate result and use it as a stepping-stone towards uncovering a set of SCM specific metrics.

Establishing a model for calculating the ROI, as well as finding SCM specific metrics and benchmarks, is something that calls for both theoretical analysis and empirical validation. However, as it was also pointed out by the previous work of Larsen and Roald [LR98], this is something that requires a long period of time and this ongoing work is currently entering the phase of the empirical validation, so in this paper, we can present only the theoretical analysis.

In the following, we first present the ROI model for SCM and the analysis that led to it, then we outline how we intend to use the coming validation of this model to identify a set of SCM specific metrics and benchmarks, and finally we draw our conclusions.

2 Step One – A Return On Investment Model

To get the broadest possible coverage of potential costs and benefits, we wanted to analyse SCM from several different perspectives. The first – and most obvious – was to analyse costs and benefits of the four canonical activities of SCM, as laid out by standards like ISO 10007:2004 and ANSI/EIA-649: configuration identification, configuration control, configuration status accounting and configuration audit. To get a better idea of the extent of especially the benefits, we wanted to widen our analysis and focus not only on the coding phase of software development, but cover the entire life cycle from requirements analysis through to maintenance. Finally, we wanted to look at the people involved with SCM tasks to be sure to get a complete coverage of benefits and – in particular – costs.

In the following, we briefly describe the results of the analysis from each of the three perspectives. This is followed by the presentation of the model that results from these analyses.

2.1 Analysis by SCM activities

In general, SCM is looked at as a management discipline that can help in planning and running a project. As such, it provides a SCM plan that describes the processes that have to be followed and a tool – or set of tools – that to some degree can automate some of these processes. The general costs that we get are those associated with the tool(s) (licences, maintenance), the making and maintenance of SCM plans, and the training of people to understand and follow the established processes and tool(s). To know more about the benefits we treat each of the four activities in turn.

Configuration identification. This activity deals with the recording and communication of information of identified configuration items and the baselining of these items. It establishes clear and effective naming conventions and how to hierarchically structure sets of configuration items. This creates a clear navigational structure for information retrieval and explicitly addresses traceability in different contexts.

Configuration control. Most important here are the change process and the Change Control Board (CCB) that defines how changes are handled. This ensures that the impact of changes is assessed and analysed, and that changes are planned and managed. This allows us to trace the status of changes as well as the entire product through its various baselines.

Configuration status accounting. This is a system of formatted reports created based on the data available in the SCM system. The associated cost is that of creating the original data, the benefits those of traceability and better information for management.

Configuration audit. Audits verify the functional characteristics and the form and fit of the product. The costs are those of carrying out the audits, the benefits are improved stability and quality of baselines/releases.

2.2 Analysis by the product's life cycle

Usually SCM is considered an activity that is aimed only at the coding phase of a software development project. However, all phases can and should apply the configuration management principles and methods to their work products. We will look at each of the life cycle phases in turn.

Requirements. By applying SCM to the requirements phase products we can get the same benefits (and costs) as for the Coding phase. Furthermore, we can obtain traceability between the work products of the different phases.

Design. This phase is equivalent to the Requirements phase.

Coding. The costs are getting tool(s), processes and training in place. Benefits come from being able to handle variations, reused code, work in parallel (and/or distributed), automated builds and facilitating the co-ordination and collaboration between programmers and teams. Furthermore, traceability, both between phases and between versions (or variants), is a benefit.

Testing. Testers benefit from the use of documented baselines to create the builds that are to be tested. This way is always clear exactly what is being tested.

Release. The use of documented baselines makes is possible to always recreate old releases. Furthermore, configuration audits ensures the quality of the release.

Maintenance. This phase probably has the most benefits from SCM. From the "information base" that SCM provides we obtain traceability and history that helps in tracking down and removing bugs. Benefits when adding new functionality are the same as for the coding phase.

2.3 Analysis by people involved

In the analysis so far, we have mostly looked at SCM from the company's point of view. In this section, we acknowledge that there are different roles that are related to using SCM and that these roles have different interests, as is also pointed out by Hass [Hass03].

Senior management. They share many interests and benefits with the company – better quality of the product, faster and more flexible response to customer requests and shorter time-to-market times.

Project management. Most benefits are related to the planning, scheduling and managing of activities. Status accounting gives improved insight into the project status and the use of a CCB ensures that all changes are controlled and impact analysed so they can finish in time and within budget.

Developer. As pointed out by Babich [Babich86], SCM is just as much about team collaboration and co-ordination as it is about management. In our case, we use the term developer in a very broad sense, as also people from other phases of the life cycle can use SCM for "developing" their products. The concepts of baseline and workspace allows for stability in the developer's daily work. The flexible merging of parallel work improves productivity, as people (or projects) do not have to wait for each other. The possibility to return to an older version gives courage and security. Traceability and the ability to highlight differences between two versions is a great help both during development and in particular during maintenance (debugging).

SCM manager. The fact that this role exists, indicate that there are also personnel costs from SCM. Someone has to take care of SCM plans and processes. Usually the SCM manager also takes part in the CCB meetings as secretary.

2.4 The resulting model

We are now ready to put all the pieces of costs and benefits together. However, before we can do that, we have to deal with a problem that we mentioned in the Introduction – the fact that that many costs and benefits are subjective or/and hard to quantify.

The aspect of subjectivity has to do with the fact that often we cannot exclude the specific context in which things are carried out. For instance, the degree of benefit we have from being able to handle variants depends on the degree to which we actually have to deal with variant products. Another factor that can cause problems is that it can be difficult or impossible to quantify a certain benefit or cost. For instance it would be difficult to quantify the actual advantage of getting to the market two months earlier than your competition – in some cases it would be a matter of survival of the company and as such the parameter should have an infinitely high value.

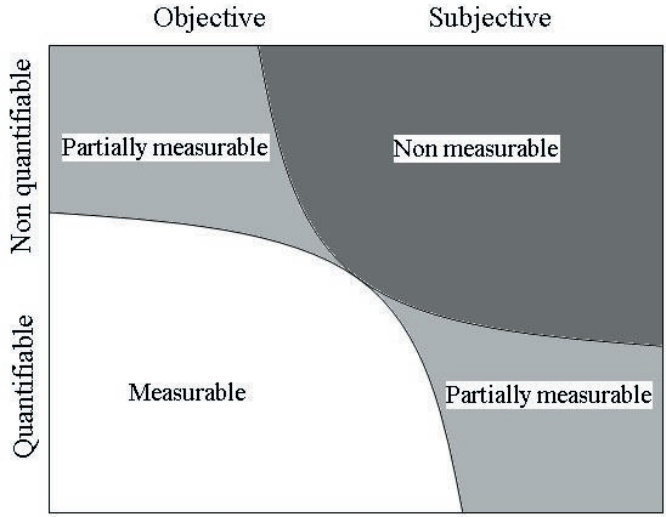


Figure 1. Characterising the measurability of parameters

To reflect these difficulties, we have divided up the parameters into three categories: measurable, partially measurable and not measurable, as shown in Figure 1. Our intention with this categorization is that the measurable parameters can be used in a general, basic ROI model, where the values of the parameters can be taken from common benchmarks and will vary very little from company to company. The partially measurable parameters will not apply to all companies and will have values that can vary wildly from company to company and as such has to be estimated by each company in question. Finally, the not measurable parameters will have to be considered an added bonus to the result from actually calculating a ROI – they do not apply to all companies and it does not make sense to try to estimate values for these parameters.

Using this categorization of costs and benefits, we obtain the two-by-three matrix of parameters shown below in Figure 2. We can see that for the costs, most are measurable, whereas for the benefits most are not measurable. This goes for the number of costs and benefits, not necessarily for their economical impact.

It is not a good property for a ROI model to have a high number of parameters that are not measurable. However, our model is still preliminary and we hope to be able to move parameters towards being measurable as we work with the empirical validation of the model. Furthermore, we can see that for the cost side most parameters are measurable whereas for the benefit side most parameters are not measurable. This means that if we use only measurable and partially measurable parameters to calculate the return on investment, we can be quite sure not to get unpleasant surprises from "hidden" costs and expect even more benefits in addition to the calculated ROI.

The model we present in figure 2 shows only an overview of the parameters. Obviously many details will be needed about the parameters to clarify their exact nature and definition. This can be done [Borracci05]; however, we leave it out here for reasons of space.

	Measurable	Partially measurable	Not measurable

Costs	• Tool costs
	• Licenses and maintenance costs
	• Training costs (cost of training the system administrator and the developers to the tool and the new SCM processes
	• Added work associated with new SCM tasks for the:
	  ○ Config. Manager
	  ○ Admin &
	  ○ Developer technician
		• Change process may be more complicated (loss of time and money waiting for authorization, average CCB-time)
		• Loss of time for doing the status accounting reports (depending on the tool and the level of automation)
			• Fear of new procedures

Benefits	• Decrease of the externally reported defects (defect report arrival rate) • Less time per bugfix • Ability to trace the original product through its development • Save time with automated software builds • Manage versions, parallel work, automatic merges • Traceability implies less time for V&V and testing	• Decrease of number of staff changes / help to integrate new employees (less cost of training a new employee) • Allows to handle very complex activities (variation of a product) • Reusing existing code and reducing repetitive development efforts • Gain factor fixing bugs in different variants • Helps the maintenance • Changes are planned, their impact is assessed • Reducing the number of errors	• Employees are happier • Provides for communication and coordination in the group • Pleasure of working in stability with a baseline and an own workspace • Working from home and distributed development • Ability to bring out the product earlier • Decrease the time required to respond to user requests • Assures that the customer gets what he paid for • Audit at the end of each phase assures consistency of the work • Provides visibility of the project • Achieves a sense of organisation and control instead of chaos
	Quantitative ↔ Qualitative		

Figure 2. The Return-On-Investment model

We have not stated an explicit formula to calculate the ROI in this paper, but have left it at the model showing the parameters. Such a formula can be made [Borracci05], but it becomes very complex and is probably of little use. What complicates matters the most is that some costs and benefits are one-time (like buying the tool) or once a year (like licenses), whereas others are daily (like the support for parallel work). Yet others are not linear (low benefit until you get to know the tool/process). Therefore it

is difficult to make a precise calculation that takes into account all these factors. However, the previous study of Larsen and Roald [LR98] indicates that the ROI in SCM is high enough that such a precise calculation should not be necessary to justify the introduction of SCM tools and processes. When it comes to upgrading tools and/or processes, the number of relevant parameters will probably be low enough to allow a precise calculation. However, in many cases the most interesting will not be to calculate the profitability of SCM, but rather to estimate the SCM costs of a project as the tool(s) and processes are already given by the company standard.

In the present model, we have not considered such parameters as compliance requirements and support of development methods. We believe that these are political issues that will not be influenced by SCM economics nor by the ability or not of SCM to support them – we may, though, be wrong about this.

3 Step Two – Looking for Metrics

Now that we have a model for the costs and benefits of SCM, we can use that model and its parameters to arrive at what we are really looking for – a set of metrics that are targeted specifically at measuring the SCM processes.

The project information base, that the SCM repository constitutes, is an obvious source of data for metrics. In fact the Configuration Status Accounting activity is mostly concerned with putting together data that can be used to monitor and manage projects and processes. However, these metrics are mostly targeted at general software engineering processes and very rarely are the SCM processes ever considered. We know of only the two cases of Farah [Farah04] and Jönsson [Jönsson04] and feel that the field needs more work to advance it to a more mature and complete state.

Just like other processes, SCM processes need to be kept an eye on and to be improved. This cannot be done if we do not have data from a set of SCM specific metrics that can be used to measure the performance of the SCM processes. We need information about the current state to make data-driven decisions about changes. And we need to track our progress to be able to assess the impact of SCM process changes.

In a project course at our department [HBM05], teams have to do four releases during the course of six XP-iterations (each iteration being 14 hours of work). We keep an explicit SCM metric for the time it takes to produce a release (extract code from the repository, compile it and carry out unit and acceptance tests, and put together system, source code, manual and documentation in one package ready to ship to the customer). The first release is done manually in 2-4 hours, teams improve for each release and most teams have an automated forth release – with a record to beat of 38 seconds. The customers guarantee that the teams do not trade quality for speed.

A set of SCM metrics and associated benchmarks can also be used to predict SCM costs for new projects drawing on data from old and current projects. But we need to find and define a set of SCM metrics and to collect data. The SCM metrics will tell us when our SCM processes are working as expected – and more importantly, the anomalies will give us early warning about SCM processes with potential problems.

In parallel with the project where we validate the ROI model, we also intend to establish a tentative set of SCM metrics. During a longer time-span we want to experi-

ment with those metrics to see what stories they can tell about the company's SCM processes and a possible change of tool. We do not expect all the parameters from our ROI model to become useful SCM metrics – and we expect more to pop up.

Already now we have some ideas for what metrics could be used for improving the SCM processes. The time to produce a release, as mentioned above, is just one. Others could be: number of merge conflicts, time to do a configuration audit, accuracy of impact analysis – and many more. However, we still need to do a lot of work here and would like to discuss our preliminary findings and our ideas for the continued work in a forum of experts.

4 Concluding Remarks

We have screened our proposed ROI model with the local branch of a global company. They have found it useful and want to adopt it, not for calculating the ROI of introducing SCM as they already has that in place, but to evaluate the profitability of proposed changes to their SCM tool and/or processes.

During the course of the model's validation, we want to look for a set of SCM specific metrics to help manage and tune SCM processes. The empirical validation of the ROI model will surely show that some parameters have only marginal effect and are best left out in order to reduce the model's complexity. We also expect new parameters to emerge that we have overlooked. And finally, it is our hope that with practical experience, we can move some of the parameters towards being more measurable.

The ROI model that we have presented is just to be considered an intermediate step; our ultimate goal is to uncover a set of "pure" SCM specific metrics. This is work in progress that we want to evaluate and discuss now that we are in the transition from phase one (the ROI model) to phase two (the SCM metrics).

References

[Babich86]: Wayne A. Babich: *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley Publishing Company, 1986

[Borracci05]: Lorenzo Borracci: *A Return on Investment Model for Software Configuration Management*, Masters Dissertation, Lund Institute of Technology, May 2005.

[Farah04]: Joe Farah: *Metrics and Process Maturity*, The Configuration Management Journal, December 2004.

[Hass03]: Anne Mette Jonassen Hass: *Configuration Management Principles and Practice*, Addison-Wesley Publishing Company, 2003.

[HBM05]: Görel Hedin, Lars Bendix, Boris Magnusson: *Teaching eXtreme Programming to Large Groups of Students*, Journal of Systems and Software, January 2005.

[Jönsson04]: Henrik Jönsson: *Graphs for Change Requests*, The Configuration Management Journal, December 2004.

[LR98]: Jens-Otto Larsen, Helge M. Roald: *Introducing ClearCase as a Process Improvement Experiment*, in proceedings of the SCM-8 Symposium, Brussels, Belgium, 1998.

[Merant01]: *Assessing Return on Investment for Enterprise Change Management Systems*, Merant White Paper, 2001.

Service Configuration Management

Eelco Dolstra, Martin Bravenboer, and Eelco Visser

Department of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands {eelco,martin,visser}@cs.uu.nl

Abstract. The deployment of services — sets of running programs that provide some useful facility on a system or network — is typically implemented through a manual, time-consuming and error-prone process. For instance, system administrators must deploy the necessary software components, edit configuration files, start or stop processes, and so on. This is often done in an *ad hoc* style with no reproducibility, violating proper configuration management practices. In this paper we show that build management, software deployment and service deployment can be integrated into a single formalism. We do this in the context of the Nix software deployment system, and show that its advantages — co-existence of versions and variants, atomic upgrades and rollbacks, and component closure — extend naturally to service deployment. The approach also elegantly extends to distributed services. In addition, we show that the Nix expression language can simplify the implementation of crosscutting variation points in services.

1 Introduction

The deployment of software services — sets of running programs that provide some useful facility on a system or network — is very often a time-consuming and error-prone activity for developers and system administrators. In order to produce a working service, one must typically install a large set of components, put them in the right locations, write configuration files, create state directories or initialise databases, make sure that all sorts of processes are started in the right order on the right machines, and so on. These activities are quite often performed manually, or scripted in an *ad hoc* way.

A software service typically consists of a set of processes running on one or more machines that cooperate to provide a useful facility to an end-user or to another software system. An example might be a bug tracking service, implemented through a web server running certain web applications, and a back-end database storing persistent data. A service generally consists of a set of software components, static data files, dynamic state (such as databases and log files), and configuration files that tie all these together.

A particularly troubling aspect of common service deployment practice is the lack of good *configuration management*. For instance, the software environment of a machine may be under the control of a package manager, and the configuration files and static data of the service under the control of a version management system. That is, these two parts of the service are both under CM control. However, the *combination* of the two *isn't*: we don't have a way to express that (say) the configuration of a web server consists of a composition of a specific instance of Apache, specific versions of configuration files and data files, and so on.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

This means that we lack important features of good CM practice. There is no *identification*: we do not have a way to name what the running configuration on a system is. We may have a way to identify the configurations of the code and data bits (e.g., through package management and version management tools), but we have no handle on the composition. Likewise, there is no *derivation management*: a software service is ideally an automatically constructed derivate of code and data artifacts, meaning that we can always automatically rebuild it, e.g., to move it to another machine. However, if administrators ever manually edit a file of a running service, we lose this property.

In practice, we see that important service deployment operations are quite hard. Moving a service to another machine can be time-consuming if we have to figure out exactly what software components to install to establish the proper environment for the service. Having multiple instances of a service (e.g., a test and production instance) running side-by-side on the same machine is difficult since we must arrange for the instances not to overwrite each other, which often entails manually copying files and tweaking paths in configuration files. Performing a roll-back of a configuration after an upgrade might be very difficult, in particular if software components were replaced.

In this paper we argue that we can overcome these problems by integrating build management, software deployment, and service deployment into a single formalism. We do this in the context of the *Nix deployment system* [8, 7], which we developed to overcome a number of problems in the field of software deployment, such as the difficulty in reliably identifying component dependencies and in deploying versions of components side-by-side. We show in this paper that the Nix framework is well suited for service deployment, simply by treating the static, non-code parts of service configurations as components. By doing so, we can build, deploy and manage services with the same techniques that we previously applied to component deployment.

The central contribution of this paper is that we posit that deployment and service configuration can be integrated by viewing service configurations as components. This allows all of Nix's advantages to apply to the service configuration realm. Specifically, this yields the following contributions:

- Services become *closures*. Since Nix helps to prevent undeclared dependencies, we can be reasonably certain that the *Nix expression* that describes a service has no external dependencies, i.e., is self-contained. Thus, we can unambiguously reproduce such a service. For instance, we can trivially pick it up and move it to another machine, and it will still work.
- Different instances of a service can automatically co-exist on the same machine. This includes not just variation in feature space (e.g., test and production instances), but also variation in time (e.g., older versions of the service). This allows us to efficiently and reliably roll back to previous configurations.
- Since deployment of code and non-code parts of a service are now integrated in a single formalism and tool, deployment effort and complexity is substantially reduced.
- Nix's functional expression language allows variability in configurations to be expressed succinctly. A major annoyance in writing configuration files for services is a *lack of abstraction*: crosscutting configuration choices (e.g., a port number) often end up in many places in many different configuration files and scripts. By gener-

ating these from feature specifications specified in Nix expressions, we reduce the deployment effort and the opportunity for errors due to inconsistencies.
- We show how services can be composed from smaller sub-services, rather than having a single monolithic description of the service as a whole.
- Multi-machine, multi-platform configurations can be described succinctly because Nix expressions can specify for what platform each component is to be built, and on what machine it is to run. Thus we obtain a centralised view of a distributed service.

Outline This paper is structured as follows. In Section 2 we motivate our work, present a simple, 'monolithic' example of service deployment through Nix, and introduce the underlying concepts and techniques. Section 3 shows how Nix services can be made compositional. In Section 4 we show how crosscutting configuration aspects can be implemented. We demonstrate how distributed deployment can be elegantly expressed in Nix in Section 5. We discuss our experiences with Nix service deployment in Section 6, and related work in Section 7.

2 Overview

2.1 Service components

From the perspective of a user, a service is a collection of data in some format with a coherent set of operations (use cases) on those data. Typical examples are a Subversion version management service [12] in which the data are repositories and the operations are version management actions such as committing, adding, and renaming files; a TWiki service [2] in which the data are web pages and operations are viewing, editing, and adding pages; and a JIRA issue-tracking service in which the data consists of entries in an issue data-base and the operations are issue management actions such as opening, updating, and closing issues. In these examples, the service has persistent data that is stored on the server and that can be modified by the operations of the server. However, a service can also be *stateless* and just provide some computation on data provided by the client at each request, e.g., a translation service that translates a document uploaded by the client.

From the perspective of a system administrator, a service consists of *data directories* containing the persistent state, *software components* that implement the operations on the data, and a *configuration* of those components binding the software to the data and to the local environment. (For the purposes of this paper, we define a component as a unit of deployment that can be automatically built and composed.) Typically, the configuration includes definitions of paths to data directories and software components, and items such as the URLs and ports at which the service is provided. Furthermore, the configuration provides *meta-operations* for initialising, starting, and stopping the service. **Fig. 1** illustrates this with a diagram sketching the composition of a version management service based on Apache and Subversion components. The *control* component is a script that implements the meta-operations, while the file `httpd.conf` defines the configuration. The software components underlying a service are generally not self-contained, but rather composed from a (large) number of auxiliary components. For

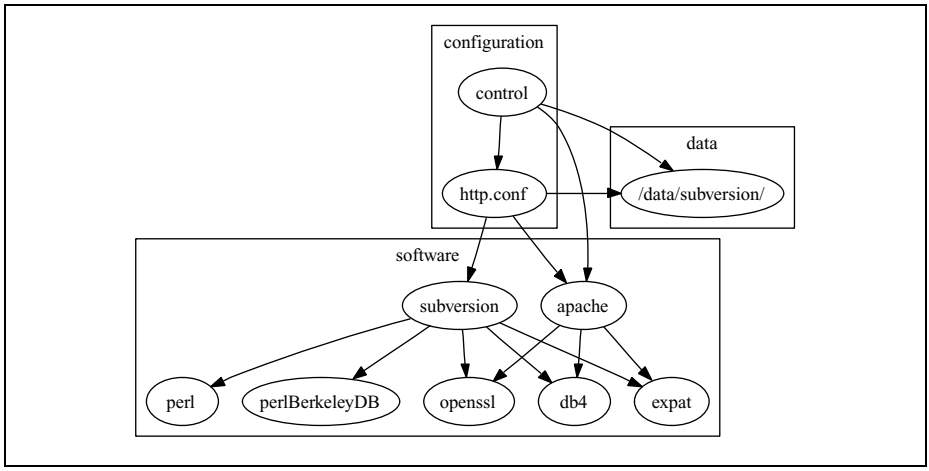


Fig. 1. Dependencies between code, configuration, and data components in a subversion service.

```
ServerRoot "/var/httpd"
ServerAdmin eelco@cs.uu.nl
ServerName svn.cs.uu.nl:8080
DocumentRoot "/var/httpd/root"
LoadModule dav_svn_module /usr/lib/modules/mod_dav_svn.so
<Location /repos>
    AuthType Basic
    AuthDBMUserFile /data/subversion/db/svn-users
    ...
    DAV svn
    SVNParentPath /data/subversion/repos
</Location>
```

Fig. 2. Service configuration. Excerpts from a `httpd.conf` file showing hosting of Subversion by an Apache server.

example, **Fig. 1** shows that Subversion and Apache depend on a number of other software components such as OpenSSL and Berkeley DB.

2.2 Service configuration and deployment

Setting up a service requires installing the software components and all of their dependencies, ensuring that the versions of the installed components are compatible with each other. The data directories should be initialised to the required format and possibly filled with an initial data set. Finally, a configuration file should be created to point to the software and data components. For example, **Fig. 2** shows an excerpt of an Apache `httpd.conf` configuration file for a Subversion service. The file uses absolute pathnames to software components such as a WebDAV module for Subversion, and data directories such as the place where repositories are stored.

Installation of software components using traditional package management systems is fraught with problems. Package managers do not enforce the completeness of dependency declarations of a component. The fact that a component works in a test environment, does not guarantee that it will work on a client site, since the test environment may provide components that are not explicitly declared as dependencies. As a consequence, component compositions are not reproducible. The installation of components in a common directory makes it hard to install multiple versions of a component or to roll back to a previous configuration if it turns out that upgrading produces a faulty configuration.

These problems are compounded in the case of service management. Typically, management of configuration files is not coupled to management of software components. Configuration files are maintained in some specific directory in the file system and changing a configuration is a destructive operation. Even if the configuration files are under version management, there is no coupling to the versions of the software components that they configure. Thus, even if it is possible to do a roll back of the configuration files to a previous time, the installation of the software components may have changed in the meantime and become out of synch with the old configuration files. Finally, having a global configuration makes it hard to have multiple versions of a service running side by side, for instance a production version and a version for testing an upgrade.

2.3 Capturing component compositions with Nix expressions

The Nix software deployment system was developed to deal with the problems of correctness and variability in software deployment. Here we show that Nix can be applied to the deployment of services as well, treating configuration and software components uniformly. We first show the high-level definition of service components and their composition using Nix expressions and then discuss the underlying implementation techniques.

The Nix expression language is a simple functional language that is used to define the derivation of software components from their dependencies. (A more detailed exposition of the language is given in [7].) The basic values of the language are strings, *paths* such as ./builder.sh, and *attribute sets* of the form $\{f_1{=}e_1;\ldots f_n{=}e_n;\}$, binding the value of expressions e_i to fields f_i. The form $\{f_1,\ldots,f_n\}$: e defines a *function* with body e that takes as argument an attribute set with fields named f_1, ..., f_n. A function call $e_1\,e_2$ calls the function e_1 with arguments specified in the attribute set e_2.

For example, **Fig. 3** shows the definition of a function for building a Subversion component given its build-time dependencies. The keyword inherit in attribute sets causes values to be inherited from the surrounding scope. The dependency stdenv comprises a collection of standard utilities for building software components, including a C compiler and library. The function fetchurl downloads a file given its URL and MD5 hash. Likewise, the other arguments represent components such as OpenSSL. Thus, the body of the function is an attribute set with everything needed for building Subversion. The build script builder.sh (**Fig. 4**) defines how to build the component given its inputs, in this case, using a typical command sequence for Unix components. The locations of the build inputs in the file system are provided to the script using environment variables.

```
{ stdenv, fetchurl, openssl, httpd, db4, expat }:
stdenv.mkDerivation {
  name = "subversion-1.2.0";
  builder = ./builder.sh;
  src = fetchurl {
   url = http://.../subversion-1.2.0.tar.bz2;
   md5 = "f25c0c884201f411e99a6cb6c25529ff";
  };
  inherit openssl httpd expat db4;
}
```

Fig. 3. subversion/default.nix: Nix expression defining a function that derives a Subversion component from its sources and dependencies.

```
tar xvfj $src
cd subversion-*
./configure --prefix=$out --with-ssl --with-libs=$openssl ...
make
make install
```

Fig. 4. subversion/builder.sh: Build script for the Subversion component.

The special environment variable out indicates the location where the result of the build action should be installed. An instance of the Subversion component can be created by calling the function with concrete values for its dependencies. For example, assuming that pkgs/i686-linux.nix defines a specific instance of the Subversion function, the invocation

```
nix-env -f pkgs/i686-linux.nix -i subversion
```

of the command nix-env builds this instance and makes it available in the environment of the user. The result is a component composition that can be uniquely identified and reproduced, as will be discussed below.

A service can be constructed just like a software component by composing the required software, configuration, and control components. For example, **Fig. 5** defines a function for producing a Subversion service, given Apache (httpd) and Subversion components. The build script of the service creates the Apache configuration file httpd.conf and the control script bin/control from templates by filling in the paths to the appropriate software components. That is, the template httpd.conf.in contains placeholders such as

```
LoadModule dav_svn_module @subversion@/modules/mod_dav_svn.so
```

rather than absolute paths. The funcion in **Fig. 5** can be instantiated to create a concrete instance of the service. For example, **Fig. 6** defines the composition of a concrete Subversion service using a ./httpd.conf file defining the particulars of the service. Just like installing a software component composition, service composition can be reproducibly installed using the nix-env command:

```
{ stdenv, apacheHttpd, subversion }:
stdenv.mkDerivation {
  name    = "svn-service";
  builder = ./builder.sh; # Build script.
  control = ./control.in; # Control script template.
  conf    = ./httpd.conf.in; # Apache configuration template.
  inherit apacheHttpd subversion;
}
```

Fig. 5. services/svn.nix: Function for creating an Apache-based Subversion service.

```
pkgs = import ../pkgs/system/all-packages.nix;
subversion = import ../subversion/ {
  # Get dependencies from all-packages.nix.
  inherit (pkgs) stdenv fetchurl openssl httpd ...;
};
webServer = import ../services/svn.nix {
  inherit (pkgs) stdenv apacheHttpd;
};
```

Fig. 6. configuration/svn.nix: Composition of a Subversion service.

```
nix-env -p /nix/profiles/svn -f configuration/svn.nix -i
```

The service installation is made available through a *profile*, e.g., /nix/profiles/svn, which is a symbolic link to the composition just created. The execution of the service, initialisation, activation, and deactivation, can be controlled using the control script with commands such as

```
/nix/profiles/svn/bin/control start
```

2.4 Maintenance

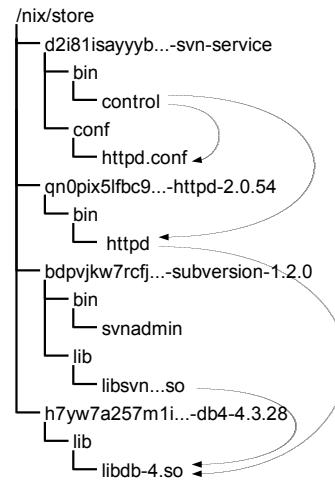
A service evolves over time. New versions of components become available with new functionality or security fixes; the configuration needs to be adapted to reflect changing requirements or changes in the environment; the machine that hosts the service needs to be rebooted; or the machine is retired and the service needs to be migrated to another machine. Such changes can be expressed by adapting the Nix expressions appropriately and rebuilding the service. The upgrade-server script in **Fig. 7** implements a common sequence of actions to upgrade a service: build the new service, stop the old service, and start the new one. (In Section 6.2 we sketch how we can prevent the downtime between stopping and starting the old and new configurations.) Since changes are non-destructive, it is possible (through a similar command sequence) to roll back to a previous installation if the new one is not satisfactory. By keeping the Nix expressions defining a service under version management, the complete history of all aspects of the service is managed, and any version can be reproduced at any time. An even better

```
# Recall the old server
oldServer=$(readlink -f $profiles/$serverName || true)

# Build and install the new server.
nix-env -p $profiles/$serverName -f "$nixExpr" -i

# Stop the old server.
if test -n "$oldServer"; then
    $oldServer/bin/control stop || true
fi

# Start the new server.
$profiles/$serverName/bin/control start
```

Fig. 7. upgrade-server script: Upgrading coupled to activation and deactivation.

approach is to have upgrade-server build directly from a version management repository, rather than from a working copy. That way, we can always trace a running service configuration back to its sources in the version management system.

2.5 Implementation



Nix supports side-by-side deployment of configurations and roll-backs by storing components in isolation from each other in a *component store* [8]. An example for some of the components built from the webServer value in **Fig. 6** is shown on the right. The arrows indicate references (by file name) between components. Each call to mkDerivation results in the construction of a component in the Nix store, residing in a path name containing a *cryptographic hash* of all inputs (dependencies) of the derivation. That is, when we build a derivation described in a Nix expression, Nix recursively builds its inputs, computes a path for the component by hashing the inputs, then runs its builder to produce the component. Due to the hashing scheme, changing any input in the Nix expression and rebuilding causes the new component to end up in a different path in the store. Hence, different components never overwrite each other. The advantage of hashes is that they prevent undeclared build-time dependencies, and enable detection of runtime dependencies by scanning for hashes in the files of components.

By storing components in isolation, Nix enables side by side installation of services. That is, there can be multiple configuration files instantiating the same software components for different services. For example, two Subversion servers with different authentication regimes, or webservers for different domains can be hosted on the

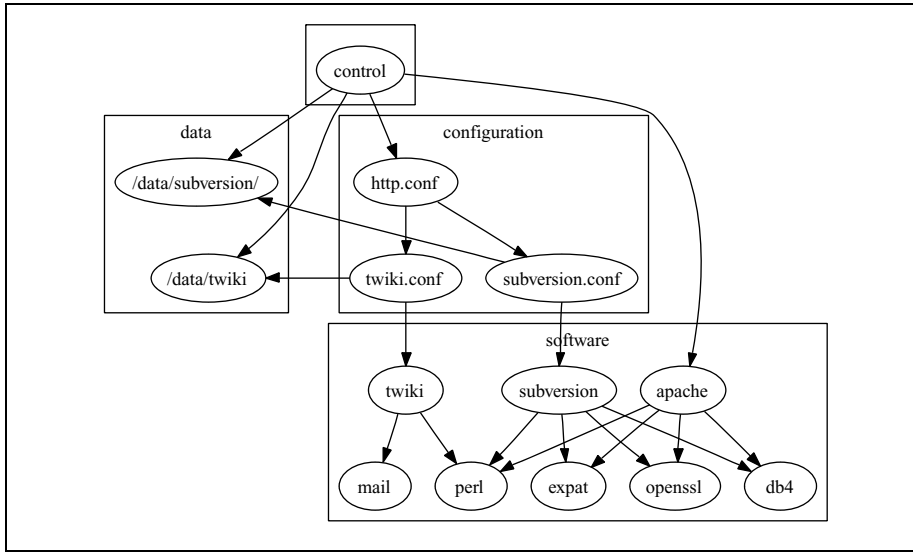


Fig. 8. Architecture of an Apache-based composition of the Subversion and TWiki services.

same machine. Furthermore, since a collection of Nix expressions exactly describes all aspects of a service, it is easy to reproduce service installations on multiple machines.

We support roll-backs through profiles, which as mentioned above are symbolic links to the current instance of a service configuration. When we build a new configuration, we make the symlink point to the store path of the new configuration component. To perform a roll-back, we simply switch the symlink back to the store path of the previous configuration.

3 Composition of Services

In the previous section we have shown a sketch of an Apache-based Subversion service deployed using Nix. The major advantage is that such a service is a closure: all code and configuration elements of the service are described in the Nix expression, and can thus be reproduced at any time. Clearly, we can deploy other services along similar lines. For instance, we might want to deploy an Apache-based TWiki service by providing Nix expressions to build the TWiki components and to compose them with Apache (which entails writing a parameterised `httpd.conf` that enables the TWiki CGI scripts in Apache). However, suppose we now want a *single* Apache server that provides both the Subversion and TWiki services. How can we easily *compose* such services?

We solve this by factoring out the service-specific parts of `httpd.conf` into separate components called *subservices*. That is, we create configuration components that contain `httpd.conf` fragments such as `twiki.conf` and `subversion.conf`. The top-level `httpd.conf` merely contains the global server configuration such as host names or port numbers, and includes the service-specific fragments. A sketch of this composition is shown in **Fig. 8**.

```
subversionService = import ../subversion-service {
  httpPort = 80; # see Section 4.
  reposDir = "/data/subversion"; ...
};
jiraService = import ../jira-service {
  twikisDir = "/data/twiki"; ...
};
webServer = import ../apache-httpd {
  inherit (pkgs) stdenv apacheHttpd;
  hostName = "svn.cs.uu.nl";
  httpPort = 80;
  subServices = [subversionService jiraService];
};
```

Fig. 9. Nix expression for the service in **Fig. 8**.

Fig. 9 shows a concrete elaboration. Here, the functions imported from ../subversion-service and ../jira-service build the Subversion and JIRA configuration fragments and store them under a subdirectory types/apache-httpd/conf/ in their prefixes. The function imported from ../apache-httpd then builds a top-level httpd.conf that includes those fragments. That is, there is a *contract* between the Apache service builder and the subservices that allows them to be composed.

The subServices argument of the function in ../apache-httpd specifies the list of service components that are to be composed. By modifying this list and running upgrade-server, we can easily enable or disable subservices.

4 Variability and crosscutting configuration choices

A major factor in the difficulty of deploying services is that many configuration choices are *crosscutting*: the realisation of such choices affects multiple configuration files or multiple points in the same file. For instance, the Apache server in **Fig. 9** requires a TCP *port number* on which the server listens for requests, so we pass that information to the top-level webServer component. However, the user management interface of the Subversion service also needs the port number to produce URLs pointing to itself. Hence, we see that the port number is specified in two different places. In fact, Apache's configuration itself already needs the port in several places, e.g.,

```
ServerName example.org:8080
Listen 8080
<VirtualHost _default_:8080>
```

are some configuration statements that can occur concurrently in a typical httpd.conf. This leads to obvious dangers: if we update one, we can easily forget to update the other.

A related problem is that we often want to build configurations in several variants. For instance, we might want to build a server in test and production variants, with the former listening on a different port. We could make the appropriate edits to the Nix

```
{productionServer}:
let {
  port = if productionServer then 80 else 8080;
  webServer = import ./apache-httpd {
    inherit (pkgs) stdenv apacheHttpd;
    hostName = "svn.cs.uu.nl";
    httpPort = port;
    subServices = [subversionService jiraService];
  };
  subversionService = import ./subversion {
    httpPort = port;
    reposDir = "/data/subversion"; ...
  };
  jiraService = import ./jira {
    twikisDir = "/data/twiki"; ...
  };
}
```

Fig. 10. Dealing with crosscutting configuration choices

expression every time we build either a test or production variant, or maintain two copies of the Nix expression, but both are inconvenient and unsafe.

Using the Nix expression language we have the abstraction facilities to easily support possibly crosscutting variation points. **Fig. 10** shows a refinement of the Apache composition in **Fig. 9**. This Nix expression is now a *function* accepting a single boolean argument productionServer that determines whether the instance is a test or production configuration. This argument drives the value selected for the port number, which is propagated to the two components that require this information. Thus, this crosscutting parameter is specified in only one place (though implemented in two). This is a major advantage over most configuration file formats, which typically lack variables or other means of abstraction. For example, Enterprise JavaBeans deployment descriptors frequently become unwieldy due to crosscutting variation points impacting many descriptors.

It is important to note that due to the cryptographic hashing scheme (Section 2.5), building the server with different values for productionServer (or manually editing in the Nix expression any aspect such as the port attribute) yields different hashes and thus different paths. Therefore, multiple configurations automatically can exist side by side on the same machine.

5 Distributed deployment

Complex services are often composed of several subservices *running on different machines*. For instance, consider a simple scenario of a *JIRA bug tracking system*. This service consists of two separately running subservices, possibly on different machines: a Jetty servlet container, and a PostgreSQL database server. These communicate with each other through TCP connections.

Such configurations are often labourious to deploy and maintain, since we now have two machines to administer and configure. This means that administrators have to log in to multiple machines, make sure that services are started and running, and so on. That is, without sufficient automation, the deployment effort rises linearly.

In this section we show how one can implement distributed services by writing a single Nix expression that describes each subservice and the machine on which it is to run. A special *service runner component* will then take care of distributing the closure of each subservice to the appropriate machines, and remotely running their control scripts.

An interesting complication is that the various machines may be of different machine types, or may be running different operating systems. For instance, the Jetty container might be running on a Linux machine, and the PostgreSQL database on a FreeBSD machine. Nix has the ability to do distributed builds. Nix derivations must specify the *platform type* on which the derivation is to be performed:

```
derivation {
  name = "foo";
  builder = ./builder.sh;
  system = "i686-linux"; ... }
```

Usually, the system argument is omitted because it is inherited from the standard environment (stdenv). When we build such a derivation on a machine that is not of the appriate type (e.g., i686-linux), Nix can automatically forward the build to another machine of the appriate type, if one is known. This is done by recursively building the build inputs, copying the closures of the build inputs to the remote machine, invoking the builder on the remote machine, and finally copying the build result back to the local machine. Thus, the user needs not be aware of the distributed build: there is no apparent difference between a remote and local build.

Fig. 11 shows the Nix expression for the JIRA example. We have two generic services, namely PostgreSQL and Jetty. There is one concrete subservice, namely the JIRA web application. This component is plugged into both generic services as a subservice, though it provides a different interface to each (i.e., implementing different contracts). To PostgreSQL, it provides an initialisation script that creates the database and tables. To Jetty, it provides a WAR that can be loaded at a certain URI path.

The PostgreSQL service is built for FreeBSD; the other components are all built for Linux. This is accomplished by passing input packages to the builders either for FreeBSD or for Linux (e.g., inherit (pkgsFreeBSD) stdenv ...), which include the standard environment and therefore specify the system on which to build.

The two generic servers are combined into a single logical service by building a *service runner component*, which is a simple wrapper component that at build time takes a list of services, and generates a control script that starts or stops each in sequence. It also takes care of distribution by deploying the closure of each service to the machine identified by its host attribute, e.g., itchy.labs.cs.uu.nl for the Jetty service. For instance, when we run the service runner's start action, it copies each service, then executes each service's start action remotely.

An interesting point is that the Nix expression nicely deals with a crosscutting aspect of the configuration: the host names of the machines on which the services are to run. These are crosscutting because the services need to know each other's host names. In

```
# Build a Postgres server on FreeBSD.
postgresService = import ./postgresql {
  inherit (pkgsFreeBSD) stdenv postgresql;
  host = "losser.labs.cs.uu.nl"; # Machine to run on.
  dataDir = "/var/postgres/jira-data";

  subServices = [jiraService];
  allowedHosts = [jettyService.host]; # Access control.
};

# Build a Jetty container on Linux.
jettyService = import ./jetty {
  inherit (pkgsLinux) stdenv jetty j2re;
  host = "itchy.labs.cs.uu.nl"; # Machine to run on.

  # Include the JIRA web application at URI path.
  subServices = [ { path = "/jira"; war = jiraService; } ];
};

# Build a JIRA service.
jiraService = import ./jira/server-pkgs/jira/jira-war.nix {
  inherit (pkgsLinux) stdenv fetchurl ant postgresql_jdbc;
  databaseHost = postgresService.host;  # Database to use.
};

# Compose the two services.
serviceRunner = import ./runner {
  inherit (pkgsLinux) stdenv substituter;
  services = [postgresService jettyService];
};
```

Fig. 11. 2-machine distributed service

order for JIRA to access the database, the JIRA web application needs to know the host name of the database server. Conversely, the database server must allow access to the machine running the Jetty container. Here, host names are specified only once, and are propagated using expressions such as allowedHosts = [jettyService.host].

6 Discussion

6.1 Experience

We have used the Nix-based approach described in this paper to deploy a number of services, some in production use and some in education or development[1]. The production services are:

[1] The sources of the services described in this paper are available at http://svn.cs.uu.nl/repos/trace/services/trunk. Nix itself can be found at [1].

- An Apache-based Subversion server (svn.cs.uu.nl) with various extensions, such as a user and repository management system. This is essentially the service described in Section 3.
- An Apache-based TWiki server (http://www.cs.uu.nl/wiki/Center), also using the composable infrastructure of Section 3. Thus, it is easy to create an Apache server providing both the Subversion and TWiki services.
- A Jetty-based JIRA bug tracking system with a PostgreSQL database backend, as described in Section 5.

Also, Nix services were used in a Software Engineering course to allow teams of students working on the implementation of a Jetty-based web service (a Wiki) to easily build and run the service.

In all these cases, we have found that the greatest advantage of Nix service deployment is the ease with which configurations can be reproduced: if a developer wants to create a running instance of a service on his own machine, it is just a matter of a checkout of the Nix expressions and associated files, and a call to upgrade-server. Without Nix, setting up the requisite software environment for the service would be much more work: installing software, tweaking configuration files to the local machine, creating state locations, and so on. The Nix services described above are essentially "plug-and-play". Also, developer machines can be quite heterogeneous. For instance, since Nix closures are self-contained, there are no dependencies on the particulars of various Linux distributions that might be used by the developers.

The ability to easily set up a test configuration is invaluable, as it makes it fairly trivial to experiment with new configurations. The ability to reliably perform a rollback, even in the face of software upgrades, is an important safety net if testing has failed to show a problem with the new configuration.

6.2 Online upgrading

As described in Section 2.4, to upgrade a service from an old to a new configuration requires that we first run the stop action of the old service, and then the start action on the new service. However, this means that there is a time window in which the service is not available.

In the most general case, this is unavoidable. For instance, if we upgrade the Apache httpd server, then we *must* restart, since neither the C language nor Apache itself has any provisions for dynamic updating (replacing the code of an executing process). Dynamic updating is only very rarely available, so in such cases we have no choice but to restart. However, if the new configuration only differs from the old one in that the Apache configuration file or some other data file has changed, we can typically upgrade on the fly. Usually this is accomplished by modifying the files in question (e.g., editing httpd.conf), and then notifying the running service that it is to reload its configuration files (e.g., by sending it a HUP signal in Unix).

The problem is that this fits poorly in the Nix model. This is because Nix components, *such as service configuration files*, are pure — they cannot be modified after they have been built. Also, we would like to use a service's reload feature *only* if that is all we have to do. For instance, if code components were also changed, we want to

restart instead of reload. In other words, we wish to use configuration file reloading as an automatic optimisation of restarting only if it is safe to do so.

We can solve the first problem by adding a level of indirection to configuration files. Rather than having services load their configuration files directly from their location in the Nix store, e.g., /nix/store/4mm5dzlnhs20.../httpd.conf, we load them through *temporary symbolic links* that point to the actual locations, e.g., /tmp/instance-943/httpd.conf. We also keep a mapping from running services to the temporary links they are using. When a service is first started, we create this symlink. When we upgrade, upgrade-service simply change the symlink to point to the new configuration file (which is an atomic action on Unix), and signal the running server process to reload the configuration file.

We can address the second problem — performing a reload only when that is sufficient — by having upgrade-server compute the differences between the dependency graphs and contents of the old and new configurations, and only reload only when only "reloadable" parts have changed. This is a conservative approach. For instance, if two Apache configurations only differ in that their top-level store paths differ and contain differing httpd.conf files, then a reload is safe. If, on the other hand, any dependencies of the top-level store path changed, we consider a reload unsafe and restart instead.

7 Related work

We are not the first to realize that the deployment of software should be treated as part of software configuration. In particular the SWERL group at the University of Colorado at Boulder has argued for the application of software configuration management to software deployment [13], developed a framework for characterizing software deployment processes and tools [4], and the experimental system Software Dock integrating software deployment processes [11]. However, it appears that our work is unique in unifying the deployment of software and configuration components.

Nix is both a high-level build manager for components, and a deployment tool. As such it subsumes some of the tasks of both build managers (e.g., Make [9]), and deployment tools (e.g., RPM [10]).

Make is sometimes used to build various configuration files. However, Make doesn't allow side-by-side deployment, as running make overwrites the previously built configurations. Thus, rollbacks are not possible. Also, the Make language is rather primitive. In particular, since the only abstraction mechanisms are global variables and patterns, it is hard to instantiate a subservice multiple times. Build tools such as Odin [5] and Maak [6] have more functional specification languages.

Cfengine is a popular tool system administration tool [3]. A declarative description of sets of machines and the functionality they should provide is given, along with imperative actions that can realise a configuration, e.g., by rewriting configuration files in /etc. The principal downside of such a model is that it is *destructive*: it realises a configuration by overwriting the current one, which therefore disappears. Also, it is hard to predict what the result of a Cfengine run will be, since actions are typically specified as edit actions on system files, i.e., the initial state is not always specified. This is in contrast to the fully generational approach advocated here, i.e., Nix builder generate

configurations fully indepently from any previous configurations. Finally, since actions are specified with respect to fixed configuration file locations (e.g., /etc/sendmail.mc), it is not easy for multiple configurations to co-exist. In the present work, fixed paths are only used for truly mutable state such as databases and log files.

8 Conclusion

In this paper we have presented a method for service deployment based on the Nix deployment system. We have argued that its software deployment properties carry over nicely into the domain of service deployment. Thanks to Nix's cryptographic hashing property, we gain the ability to reliably recreate configurations, to have multiple instances of configurations exist side by side, and to roll back to old configurations. Nix's functional expression language gives us a way to deal with variability and crosscutting variation points in an efficient way, and to componentise services. Finally, the approach extends to distributed deployment.

References

1. Nix deployment system. http://www.cs.uu.nl/wiki/Trace/Nix, 2005.
2. Twiki—an enterprise collaboration platform. http://twiki.org/, 2005.
3. M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.
4. A. Carzaniga et al. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, April 1998.
5. G. M. Clemm. *The Odin System — An Object Manager for Extensible Software Environments*. PhD thesis, University of Colorado at Boulder, February 1986.
6. E. Dolstra. Integrating software construction and software deployment. In B. Westfechtel, editor, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, Oregon, USA, May 2003. Springer-Verlag.
7. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
8. E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.
9. S. I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.
10. E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.
11. R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, USA, May 1997.
12. C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, June 2004.
13. A. van der Hoek, R. S. Hall, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Software deployment: Extending configuration management support into the field. *Crosstalk, The Journal of Defense Software Engineering*, 11(2), February 1998.

ArchEvol: Versioning Architectural-Implementation Relationships

Eugen C. Nistor, Justin R. Erenkrantz,
Scott A. Hendrickson, and André van der Hoek

University of California, Irvine
Donald Bren School of Information and Computer Sciences
Department of Informatics
Irvine, CA 92697-3425 USA
{enistor, jerenkra, shendric, andre}@ics.uci.edu

Abstract. Previous research efforts into creating links between software architecture and its implementations have not explicitly addressed versioning. These earlier efforts have either ignored versioning entirely, created overly constraining couplings between architecture and implementation, or disregarded the need for versioning upon deployment. This situation calls for an explicit approach to versioning the architecture-implementation relationship capable of being used throughout design, implementation, and deployment. We present ArchEvol, a set of xADL 2.0 extensions, ArchStudio and Eclipse plug-ins, and Subversion guidelines for managing the architectural-implementation relationship throughout the entire software life cycle.

1 Introduction

Software architecture provides a high-level, abstract view of a system where it is easier to identify and reason about a system's main computational parts (the components), the ways through which they interact (the connectors), and their configuration [1]. In the early stages of development an architecture is typically used as a communication tool to provide understanding to other developers but it is also often analyzed to determine or ensure specific properties of the resulting system [2].

The benefits of having an architecture in place can extend beyond the initial stages if a mapping between the architectural model and the implementation can be maintained throughout development. If an architecture describes properties that are not accurate, then there is no point in analyzing the architectural model in the first place, since the results of the analysis cannot be guaranteed. But even if the consistency is initially ensured, changes in the implementation can lead to changes in architectural properties, leading to a phenomenon known as architectural erosion [1]. Architectures are described using specialized architecture description languages (ADLs), while implementations are described using programming languages. During regular development, new requirements might determine the necessity of branching the architecture and the associated

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

implementation of one or more components. In this case, we want to be able to accurately determine which versions of the component implementations belong to the initial version of the architecture and which belong to the branched version of the architecture. ArchEvol defines mappings between architectural descriptions and component implementations using a versioning infrastructure and addresses the evolution of the relationship between versions of the architecture and versions of the implementation.

Maintaining mappings between different versions of architecture and implementation is an issue that has not been addressed adequately to date. Some SCM systems try to blur the distinction between repository configuration and architectural configuration. Adele [3] and the Cedar System Modeller [4] provide consistency support between a description of a system in a module interconnection language and the underlying module implementations. However, their solution is enabled by having implementation, system description and configuration management support in the same system, and limits their development to the capabilities of the particular tools offered. Architectural description languages have evolved from module interconnection languages to include various new structural and behavioral properties, and thus need specialized tools that support their development and analysis beyond configuration descriptions. Popular configuration management approaches today can maintain versions of implementation and architecture as closed-semantics files and directories, just like they would keep any other type of software artifact, but lack the capability of explicitly maintaining a mapping between them [5][6]. Architecture-based development approaches have maintained mappings between single versions of the architecture and implementation [7][8]. Other approaches, such as MAE, have focused on versioning of the architecture, but do not integrally support the evolution of the implementation nor the mapping of architecture to implementation [9].

Our approach centers around promoting strong interconnections between the architecture, implementation, and the versioning sub-systems while still allowing their independent development. Not only are the architecture and implementation described using different languages and concepts, but there are different tools and environments that are used for creating and maintaining them. We do not propose one single tool to perform all tasks, but instead propose adopting the right tool available for the individual task and facilitating its interactions with the other tools. Furthermore, the way the tools are used and integrated commands an associated process. The links used in ArchEvol can be used to support decentralized development of architectures and component implementations by different parties. The architecture and the individual components can be developed in parallel and the common information between the two can be synchronized at certain points in time.

ArchEvol specifically builds upon three previously existing tools: ArchStudio [10], Eclipse [11], and Subversion [6]. Our solution could be applied to any combination of architectural development, source code development and configuration management systems, but the three that we chose have particular features that makes their integration easier. ArchStudio provides the facilities for managing ar-

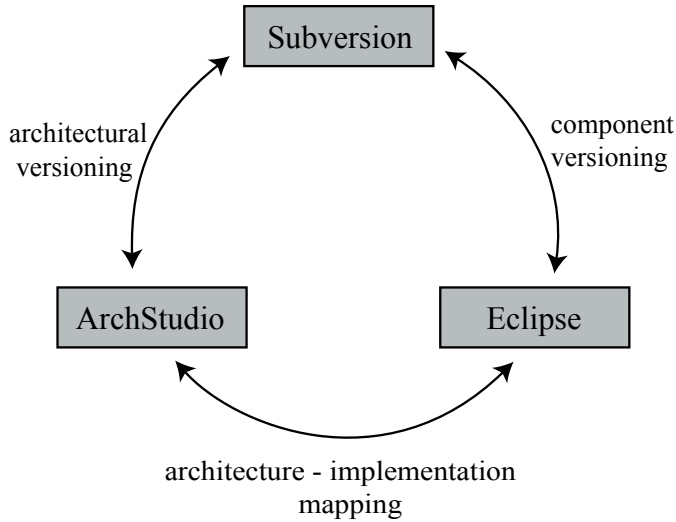


Fig. 1. ArchEvol Overview.

chitectures, through its XML-based xADL2.0 architecture description language [12]. Eclipse is an extensible development environment with comprehensive support for source code manipulation, used to manage implementations. Subversion is an SCM system that provides a WebDAV-compliant repository, and it is used to store evolving versions of the architecture and implementation. Our solution requires the creation and maintenance of a versioned set of hyperlinks between artifacts that did not previously exist, and its main contribution lies in maintaining the mapping and ensuring that the right versions of the architectural components map onto the right versions of the code (and vice versa).

2 Approach

ArchEvol provides an integrated approach for maintaining an accurate architectural model which is consistently mapped to implementation throughout the development process. This is achieved by integrating ArchStudio, an architectural environment, Eclipse, a development environment, and Subversion, a WebDAV-centric software configuration management system. While these three tools work well independently, we have created an additional layer of infrastructure and a process that allows these tools to work together to offer end-to-end automated

support for architecture-centric evolution that focuses on the identified problem of maintaining the mapping among an evolving architecture and its evolving code.

2.1 Architectural-implementation mapping

An architectural model describes properties of the components and connectors in an abstract manner. However, these architectural entities must eventually be implemented using a programming language in order to realize this abstraction. Therefore, the first step of the ArchEvol process is to integrate the ArchStudio and Eclipse environments by enabling design-time coordination between the design of the architectural model and the implementation of the components.

The integration of ArchStudio and Eclipse presented two challenges that we had to resolve in order to be successful. First, we had to create the necessary extensions to ArchStudio and Eclipse to support bi-directional communication. Secondly, and more importantly, we also had to construct a mapping between their different elements: components and connectors are the primary extensibility focus in ArchStudio, while Eclipse centers on projects, packages and classes.

Creating extensions. Both ArchStudio and Eclipse were intended to be easily extensible development environments. However, they have taken different philosophical approaches to extensibility: Eclipse allows extension through plug-ins that implement pre-defined interfaces, while ArchStudio uses loosely coupled tools that interact through exchanges of pre-defined event types. In order to have the two environments communicate, we implemented an Eclipse plug-in and an event-based ArchStudio component that are linked through an inter-process connector.

Consequently, when both environments are running, the two environments can exchange the required information about the architecture and implementation. It should be pointed out that it is not mandatory for both environments to be run together at all times. Modifications to the architecture and the implementation can be carried out independently, and when both environments are brought up together, the synchronization between the two can be completed.

Defining Mappings. Since the two environments are intended for different purposes (architecture design versus implementation), we had to devise a mapping between their elements. At the architectural level in ArchStudio, components and connectors serve as the essential building blocks of the application. In a dynamic architecture, such as the ones ArchStudio supports, these components can be added, removed, and replaced on-the-fly. However, source code deals with elements at a different level of granularity. The functionality of a component can be implemented using one or more classes. The decision of how to split this functionality is a low-level design decision that can be dictated by language restrictions, clarity of design considerations or application of design patterns. Therefore, we considered that a component should accommodate an arbitrary number of classes.

Our solution supports the notion of a component in Eclipse as a *component project*, an extension to a regular Java project. The implementation for a com-

ponent or connector in the architecture will therefore consist of the contents of the component project, and the architectural description will have links to both a source code and a binary version of the implementation. These links will point to a corresponding repository location if the component or connector is not under development, or to a local Eclipse IDE workspace if the component or connector is under development and needs to be tested before being committed. A noteworthy observation is that, in what regards the mapping to source code, we found no difference between components and connectors. The difference between them is a semantic and more important at the architectural level, but we found implementing both as a *component project* being sufficient for time being.

Besides the regular packages and java class files, this project also contains a special file that contains descriptions of component's properties as metadata [13]. Although in this paper we present the means by which to maintain the mapping between architecture and implementation consistent, an interesting problem that we plan to address in the future is how to use this mapping to enforce the semantic consistency between the properties of the implementation, as derived from the source code, and the properties claimed for the component or connector in the architectural description. The role of the metadata is to mediate this problem, and its implementation is based on a simple observation: the architectural model already contains a number of descriptions related to component implementations. In order to enable parallel development of architecture and components, the metadata in the component project will describe the same types of properties that are relevant at the architectural level.

The problem of consistency between architecture and implementation is in this way split in two parts: first, the component metadata and its implementation need to be maintained consistent, then the component metadata and the architectural model need to be synchronized. An example of such a property would be the name of a main class that is needed from the implementation in order to instantiate the component. Within Eclipse, the component metadata editor can check if the name chosen for this property actually denotes an existing class in the implementation, and will help choose a valid one if not. Then, using our integration between Eclipse and ArchStudio, the metadata can be updated into architectural models that contain the component.

This link between Eclipse and ArchStudio is important because it allows implementation for components independently from the architectural design while at the same time makes sure that the properties implemented in the components are the same as the ones described in the architecture.

2.2 Versioning Structure

Since the architectural model evolves alongside its component implementation, the architecture description should itself be versioned. Therefore, a specific version of the architecture consists of an architectural configuration made up of specific component and connector versions. Besides recording the changes to the architectural model, versioning in such a manner allows us to determine for each

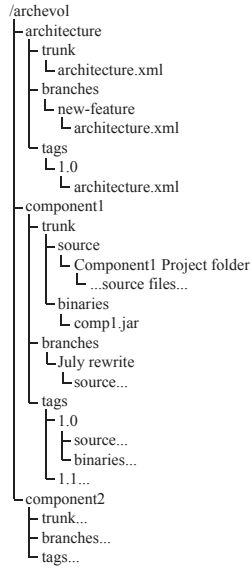


Fig. 2. ArchEvol Subversion Directory Structure.

version of the architecture the corresponding versions required for instantiation of all components and connectors.

ArchEvol intends to manage the architecture and implementation evolution throughout the design and deployment phases of the software life cycle, therefore a consistent versioning approach must be adopted throughout to ease the transition between one phase and another. This transition mandates that all necessary artifacts be consistently and uniformly available. One such technique to achieve this consistency and uniformity is to leverage a WebDAV repository to store the project artifacts.

In order to support both design-time and run-time evolution management for compiled languages (such as Java), both source files and compiled binaries need to be available at all times: our chosen method is making the resources available via a WebDAV interface. An important observation should be made here: relying upon URLs allows the source code for components and the architecture to be distributed throughout a potentially virtual organization by using multiple WebDAV servers. However, the simplistic approach would be to have one centralized WebDAV server that stores all of the necessary artifacts for a project.

Adoption of Subversion. Subversion provides a WebDAV-compliant repository used for storing project artifacts as they evolve in ArchEvol. This is a natural choice as Subversion provides a WebDAV interface as well as an Eclipse plug-in via the Subclipse client [14]. However, the adoption of Subversion does introduce one deficiency from other more traditional SCM systems in one re-

spect: a lack of per-file revision numbers. Instead, it provides a global revision number which includes directory versioning.

Therefore, for example, within Subversion, it is not possible to refer to just revision 20 of a file foo.c. Instead, you must refer to repository revision 20 which contains foo.c. To this end, it is considered standard practice by Subversion developers [15] to use a subdirectory called trunk to store the latest revisions of a project, and then copy the contents of trunk into a subdirectory within another directory called tags when an official version is to be released. Additionally, if development is to be split into multiple tracks, a corresponding copy of trunk can be copied to the branches directory where the branch can be conducted.

This directory structure means that the location within a Subversion repository explicitly indicates the version of the component, whether it is on the trunk, part of an official release tag, or part of a development branch off the mainline. As described in Figure 2, ArchEvol follows a particular versioning structure that adheres to the published Subversion best practices and satisfies the constraints of maintaining design-time and run-time evolution. The particular structure of the repository illustrates a scenario where the architecture and the implementation for the components are maintained in the same directory. However, the directory for the architecture and the directories for the components could be maintained in a distributed fashion over a number of Subversion repositories.

Versioning Components. Each component in an architecture is assigned its own directory within a Subversion repository. This directory contains two types of information: source code, containing the Eclipse project associated with the component, and binaries, where the result of compilation of the Eclipse project should be deposited.

An ArchEvol component can be opened up as a component project in Eclipse with the help of an Subversion client such as Subclipse. However, while the project is open in Eclipse, the developer has a local copy of the project that can be used to test the functionality in an open architecture in ArchStudio that uses this component. Once the changes are declared satisfactory, the project can be checked back in the trunk directory in Subversion, updating the same version, or a new version can be created and the project has to be copied in a new tag or branch directory.

In what concerns the binary packages, we now leave building them as the responsibility of the developer. The binaries have to be put in the specific directories that we prescribe, and the Subversion URL pointing to them will be synchronized between Eclipse and ArchStudio using our communication infrastructure. ArchStudio will need this information in order to instantiate the components.

One advantage of having the source code information separate from the binary packages for components is that we can provide a uniform way of instantiating the components for which the source code is not released and we only have the distribution package. Additionally, by having the release implementation for each component packaged as a jar file this offers the support for architectural deployment without a compiler. An architecture-based deployment tool will know

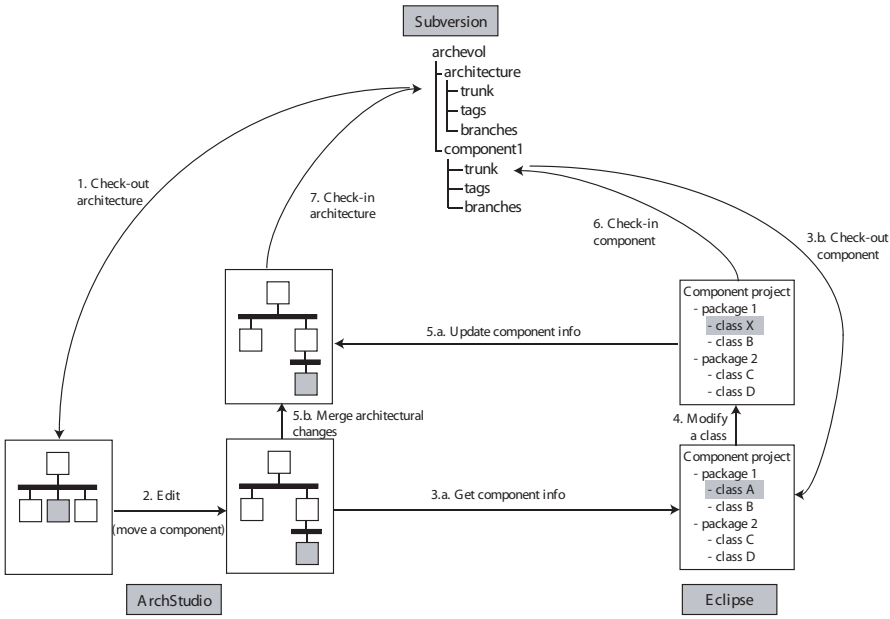


Fig. 3. ArchEvol Editing Process.

where the distribution packages are located, and it can rely upon the appropriate files from the repository.

Versioning Architectures. A version of the architectural model can be opened in ArchStudio for editing or for instantiation by providing a WebDAV URL to ArchStudio. An architectural model in ArchStudio consists of a single xADL2.0 file. The current version of the architecture will be kept in the trunk directory, while versions will be kept in tags and branches directories. It is important to point out that Figure 2 only shows the organization of directories for one architecture, but in reality the Subversion repository can contain any number of architectures for different systems.

An architectural model will contain links to the corresponding binary distribution in the WebDAV repository for each component and connector. These links can be used by ArchStudio's built-in Architecture Evolution Manager (AEM) to instantiate an architecture from the architectural description. When versioning an architecture, the links to the corresponding versions for the components are saved within the file.

3 Example

The scenario depicted in Figure 3 shows how our infrastructure helps us in fulfilling our vision for architectural-driven development and evolution. To date, the critical aspects of our process are automated, however, it is sometimes necessary to manually look up information presented in one of the three applications and then paste it into another. Our scenario notes instances where manual interaction is still necessary.

Figure 3 shows the steps involved in one cycle of development in an architecture based evolution setting. At a certain point in time, there is an architectural model and there are implementations for all the components in the architecture, and they are kept in a Subversion repository in a similar directory structure as we have described in 2.2. We want to follow the interaction of the three tools through a modification of the architecture that requires a modification of one component's implementation as well. In our scenario we move a component to a lower level in the architecture, which requires a change in the component's implementation because in the new position the component will handle additional types of messages produced by the components above it.

Step 1: In the first step, a local copy of the architecture is checked out using any Subversion client.

Step 2: The architect opens and modifies the architectural model in ArchStudio. The modification consists of adding a new connector and moving a component. Currently, ArchEvol does not determine if this type of change necessitates a change to the implementation. However, future analysis based on existing consistency mechanisms, such as incorporated in Rational Rose and other design editors to map UML to code, may be able to detect this automatically.

Step 3a: The component developer opens Eclipse which connects to ArchStudio through the ArchEvol-Eclipse plug-in. The user is presented with a list of components and connectors along with their respective source-code URLs.

Step 3b: The source-code URL is used to load the component's source code as a project into Eclipse. Currently the implementation must be manually checked out into a local folder using Subclipse. Once the implementation is checked out, a new component project is created using the contents of the checked out folder and a locally created copy of the component's architectural metadata contained in the architectural description opened in ArchStudio. It retrieves the component's architectural metadata from ArchStudio automatically. Connectors are handled in a similar way as components.

Step 4: The developer modifies the source code of the component. A consistency critic, provided by the ArchEvol plug-in, informs the developer if the class referred to in the component's architectural metadata is not present in the actual implementation.

Step 5: Once changes to the implementation are complete and consistent with the local copy of the component's architectural metadata, the local copy of the component's architectural metadata is used to replace that in architectural description. We hope to provide the ability to merge this data in the future automatically. It is not always necessary to perform this step. This is only necessary

if a change in the implementation necessitates a change in the architectural description. For example, this might occur when the name of the main class used to instantiate the component is changed. At this point, with both ArchStudio and Eclipse open at the same time, the developer can instantiate the system in order to determine if the changes to the component's source code are effective. This is possible because the URL for the binary version of the component that is being edited is temporarily replaced to point to the location of the locally compiled Java class files created by Eclipse when compiling the component's source code. Other components in the architecture that are not checked out locally can be instantiated using the binary URL provided in the architectural description. This allows a developer to instantiate an entire system without needing to check out every component within it. Steps 4, and 5 when necessary, can be repeated until the desired results are achieved.

Step 6: If the changes are satisfying, the folder for the component project in Eclipse can be committed manually using the Subclipse plug-in.

Step 7: The architectural description can be saved from ArchStudio, and the local copy can be committed using the same Subversion client that was used at step 1. The scenario described here is just one step in the development cycle. The benefits of having the versioning structure in place can also be seen during the release process. When a component's implementation is stable enough for an official release, the content of the trunk folder in the component's repository can be copied to the tags folder. The architecture can be modified so that each component's URL points only to official component releases. Likewise, it can be copied into its own tags folder. The official release architecture can then be instantiated without the need to check-out and compile any source code and without being effected by continuing development of the architecture or components.

4 Related Work

Maintaining the relationships between architectural descriptions and component implementation has been the focus of earlier research. One approach is to enforce consistency by ensuring that an implementation conforms to its architecture and that the architecture correctly represents its implementation. ArchJava [8], proposes specific Java language extensions that incorporate architectural descriptions into the source code. It enforces application communication integrity, meaning that components only communicate along architecturally declared communication channels. However, ArchJava uses implicit mappings, since architecture and implementation are described in the same files. Furthermore, ArchJava does not address any versioning constructs as the mapping between an architecture and its implementation is static.

Projects, such as Software Concordance [16], have treated implementation artifacts as hypertext with links to other project artifacts, such as specifications. The Molhado configuration management system [17], integrated with the Software Concordance environment, aims at providing a generic infrastructure

for maintaining versions of all kinds of software artifacts. In particular, MolhadoArch is very similar to ArchEvol in that it offers versioning capabilities for both the architecture and the individual components in an architecture. Their architecture-implementation mapping shares many of the same characteristics, but is specific to the Molhado environment. The architectural information can be imported from an xADL file, but by doing so it is transformed into Molhado's internal data structures. Other architectural tools such as ArchStudio cannot be used any longer on the architecture. ArchEvol, instead, builds upon the generally available architectural environment such as ArchStudio and adds integration with familiar systems of Eclipse and Subversion. Therefore, the developer experience does not change or require use of different tools.

Focus [18] takes a reverse engineering approach to mapping architecture to implementation. The source code for an application is reverse engineered to an UML class diagram and then related classes are grouped together as components. This approach maintains these mappings through evolution by applying the recovery process incrementally, to the changed portions of the source. The mapping between architecture and implementation proposed by ArchEvol is similar to the one in Focus in that a component is defined as a group of related classes. However, Focus seems more suitable for large applications that have all the source code at the same location, and where changes are first done at the implementation level. ArchEvol instead tries to promote a decoupled development model where architecture and implementation can be developed and evolved in parallel.

Traditional component-based development uses components as binary packages, but reusing them usually requires changes and customizations that can only be performed at source code level [19]. The Source Tree Composition project [20] proposes a component model that uses source code components rather than binary ones, and offers a solution to merging the source code in different components in order to build new systems with various configurations. ArchEvol uses the same definition of components as being composed out of source code elements, however Source Tree Composition goes further into solving different dependencies and building mechanisms between components. ArchEvol is focused more on the development of such components, while deployment and solving dependencies are problems that can be solved in a future extension.

5 Conclusion

Since the architecture descriptions and implementations are kept separate by ArchEvol, it is possible to have parallel development of architecture and components. We explicitly focused on integrating existing best-of-breed off-the-shelf applications to support a coherent decentralized development process. An architect can focus on working with architectural environments and tools while developers can work on implementation with familiar tools.

In the ArchEvol model, components become now not only the basic units for deployment, but also the basic units for decentralized development. Our current progress has demonstrated that ArchEvol can indeed achieve the integration

necessary to provide versioning support for maintaining relationships between architecture and implementation. While the basic functionality is present, there still remains work to increase the automation of the entire process. At present, all functionality described in this paper is possible - however, some aspects still require manual intervention where we believe automation would ultimately be possible.

We also seek to examine the long-term effects of ArchEvol on the overall development cycle and analyze whether our selected mappings between architectural entities and component projects are sufficient to describe real-world projects. Defining consistency rules between architectural description and implementation is still an open question, and involves determining which exact parts from the source code have an influence at the architectural level. With the proper feedback and refinements, we believe that ArchEvol will be able to provide developers with the support necessary to maintain meaningful relationships between architecture and implementation throughout the software life cycle.

6 Acknowledgements

Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

References

1. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17** (1992) 40–52
2. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
3. Estublier, J., Casallas, R.: The Adele configuration manager. In Tichy, W., ed.: Configuration Management. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England (1994) 99–133
4. Lampson, B.W., Schmidt, E.E.: Organizing software in a distributed environment. In: SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems, New York, NY, USA, ACM Press (1983) 1–13
5. GNU: CVS. http://www.gnu.org/software/cvs/ (2005)
6. Collabnet: Subversion. http://subversion.tigris.org (2005)
7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: 24th International Conference on Software Engineering (ICSE 2002), ACM (2002)
8. Aldrich, J., Chambers, C., Notkin, D.: Archjava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering. (2002)
9. Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae—a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13** (2004) 240–276

10. Institute for Software Research: ArchStudio 3. http://www.isr.uci.edu/projects/archstudio/ (2005)
11. Eclipse Foundation: Eclipse. http://www.eclipse.org (2005)
12. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). (2001)
13. Orso, A., Harrold, M.J., Rosenblum, D.S.: Component metadata for software engineering tasks. In: Second International Workshop on Engineering Distributed Objects (EDO 2000), Springer Verlag: Berlin (2000) 126–140
14. Collabnet: Subclipse. http://subclipse.tigris.org (2005)
15. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, M.: Version Control with Subversion. http://svnbook.red-bean.com/ (2004)
16. Nguyen, T.N., Munson, E.V. In: The software concordance: a new software document management environment. ACM Press (2003) 198–205
17. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C. In: Molhado: Object-Oriented Architectural Software Configuration Management. IEEE Computer Society (2004) 510
18. Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. Journal of Automated Software Engineering (2005)
19. Garlan, D., Allen, R., Ockerbloom, J. In: Architectural mismatch or why it's hard to build systems out of existing parts. ACM Press (1995) 179–185
20. de Jonge, M.: Source tree composition. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, Springer-Verlag (2002) 17–32

On Product Versioning for Hypertexts

Tien N. Nguyen, Cheng Thao, and Ethan V. Munson

University of Wisconsin-Milwaukee
{tien,chengt,munson}@cs.uwm.edu

Abstract. Versioned hypermedia has shown its success in promoting better understanding and management of evolving document collections in many domains. However, providing versioning capability for a hypermedia system raises several important structural and cognitive issues. Our research has produced *Molhado*, the first hypermedia infrastructure that applies the *product versioning* model to versioned hypermedia. Molhado not only supports configuration management for hypermedia structures in a fine-grained manner, but also provides version control for individual hyperlinks and document nodes. This paper explains how the product versioning model in Molhado addresses serious issues identified by earlier research on versioned hypermedia. We will also discuss the new issues raised by using this versioning model.

1 Introduction

Versioned hypermedia (or *hypertext versioning*) is concerned with storing, retrieving, and navigating prior states of a hypertext, and with allowing groups of collaborating authors to develop new states over time [1]. Hypertext versioning has shown its usefulness in several domains. In software engineering, the ability to capture the evolution of relationships between development artifacts makes versioned hypermedia more powerful than traditional versioning systems that only capture the evolution of the artifacts themselves. Relationships between requirements, design, and implementation documents can be represented directly so that traceability analysis and cost estimation become more tractable.

In legal systems, laws, regulations, and tax codes are a set of complex information artifacts with many relationships. It is important to store and retrieve previous document revisions because, in legal systems that prevent ex-post-facto laws, the version of a law that affects a case is the one in effect at the time of an infraction [1]. Hypertext support can make it easy to navigate to related laws, precedents, regulations, and codes within the set of applicable laws at the time in question. Thus legal systems can benefit from versioned hypermedia capability.

In the World Wide Web, the most popular and successful hypertext system, documents are interrelated to each other via HTML hyperlinks. Versioned hypermedia could allow users to roll a Web site back to a specific time and navigate it as if they were interacting with the Web sites as of that day. Using this facility, an E-commerce site could determine the validity of customer complaints about pricing errors or a news site could allow users to view the site for a particular day and time.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

However, providing versioning capability for a hypermedia system raises some important structural and cognitive issues, described in detail by Østerbye [2]. The structural issues are immutability of versions, version control for links, and versioning for hypermedia structure. The cognitive issues are version creation and hypermedia element version selection.

Our research has produced Molhado [3], an infrastructure for hypertext versioning and software configuration management that is well-suited for managing logical relationships among software documents. Molhado uses a single versioning mechanism for all software components and for the hypertext structures (including anchors and links) that connect them. Molhado can track changes at a very fine-grained level, allowing users to return to a consistent previous state not only of a hypertext network but also of a single node or link. Molhado supports both embedded (as in HTML) and first-class hyperlinks (as in XLink [4]). Molhado is, to the best of our knowledge, the first system that applies the *product versioning* model to versioned hypermedia. Molhado's product versioning addresses all of the issues mentioned by Østerbye. Of course, it raises other issues.

This paper is divided into seven sections, the first being this introduction. The next section presents related work and describes the most important versioning models used for hypertexts. Section 3 summarizes structural and cognitive issues with versioned hypermedia raised by Østerbye. Section 4 describes Molhado's versioned hypermedia model and explains how it addresses those issues. Section 5 presents distinguished functionality of the Molhado versioned hypermedia tool. Section 6 discusses new issues raised by the use of the product versioning model and the final section presents conclusions.

2 Related work

There are three important approaches to version control for hypertext structure and related software artifacts: the *composition model*, the *total versioning model*, and the *product versioning model*.

In the *composition model*, versions of atomic components are maintained and assembled into versions of composite components. Each atomic component (usually a file) has a version history. Versions of composite components or of the entire system are defined by revision selection rules (RSRs) that specify which component versions are part of a version of the larger artifact.

Composition-model systems that provide versioning only for data include Xanadu [5], KMS [6], DIF [7], and Hyperform [8]. The HyperWeb [9] and HyperCASE [10] systems are both based on RCS [11], in which the smallest versionable information unit is a file.

The use of RSRs allows versions of complex objects to be constructed by composition of versions of simpler objects, but it creates some problems [12]. For example, the indirect representation of a hypertext structure makes it difficult to analyze the structure, unless the rules are actually executed. So, to avoid the use of RSRs, some systems maintain a notion of *current context* [13–15]. A context has been defined as a coherent structure, such as a document, a document

collection [2], or a partition of a hypertext [13]. However, the complexity and overhead of maintaining the current context becomes significant if composite components have many nested levels and hyperlinks among them.

While the previous systems focused on versioning the linked objects, other research has explored how to maintain version histories for links as well while still using composition versioning. In Neptune [16], the composite is used to represent the isolated work area of each collaborator. HyperPro [2] records changes for links by placing links inside a "version group" (i.e., a composite) that is versioned. It supports both links to a specific version of a node and links to a node without regard to its version. Like HyperPro, HyperProp [17] does not record the history of links individually. In both HyperPro and HyperProp, the RSRs are stored on the structure, affecting all link endpoints, and provides for selection of specific document revisions from a versioned document. With this RSR approach, it is inefficient to evaluate rules across the revisions of a specific link [18].

The research addressing versioning in open hypermedia systems has also applied the composition versioning approach. Microcosm [19] has a context-like structure called the "application", which is versioned. In the versioning proposal for Chimera [20], a "configuration" is a named set of versions of Chimera hypermedia elements, representing the subset of a hypertext structure that might be affected by modifications to an externally stored object. In the hypermedia version control framework (HURL) [21], a hypertext structure is represented by an "association", which is a collection of link identifiers, anchor identifiers, and the documents that define a connection. The IAM group has investigated the Fundamental Open Hypermedia Model (FOHM)'s contextual model [22] to see if it could replace the selection engine in a hypertext versioning server.

In the *total versioning model*, all items are versioned, including composite and atomic components. Each item still has its own version space. The relationships among the items' version spaces are defined by RSRs or by versions of composites referring to versions of other components. In PCTE [23], a new version is created by recursively copying the whole composition hierarchy and establishing successor relationships between all components. In CoVer [14] and VerSE [15], the RSRs are stored on the containment arc between a container and its contents, providing selection of link revisions and document revisions from versioned links and versioned documents, respectively. This structure versioning increases the work that must be performed to ensure that a structure container holds a consistent hypertext [18]. The SEPIA [24] system has the notion of "composite node", which contains a partially ordered set of nodes and links and represents subgraphs of the hypermedia network. The total versioning model, in general, suffers from the version proliferation problem [25], where each small change in a leaf of the hierarchy creates versions of multiple objects higher in the compositional hierarchy or hyperlinked to that leaf node.

In contrast to total versioning, *product versioning* establishes a single view of a software product, or even an entire database. This is done by arranging versions of all items in a uniform, global version space. This approach is popular in software configuration management (SCM) and versioning systems such as

in change-based systems (COV [26], Aide-de-camp [27], PIE [28], etc), or state-based systems (Voodoo [29]). Wagner's product versioning infrastructure [30] integrated version management with incremental program analyses. Our system, Molhado, is currently the only versioned hypermedia system that applies this product versioning model.

3 Issues with versioning for hypertexts

This section summarizes structural and cognitive issues with providing the versioning capability for a hypermedia system mentioned by Østerbye [2].

3.1 Immutability of versions

The first structural issue is called *immutability of versions*. It is obviously that the contents of a frozen version of a resource should be immutable, that is, they cannot be changed without creating a new version of the resource; it is less clear how links pointing to it and its attributes should be treated. 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 carrying semantics or substantial parts of the resource itself, and thus their modification should definitely require the creation of a new version of the whole resource.

3.2 Version control for links

The second structural issue is associated with version control for links. Versioning for links is very important. The fact that existing hypermedia systems for software development do not version individual links is a significant factor preventing their wider use in the software engineering domain [1]. However, providing version control for individual links raises some serious problems.

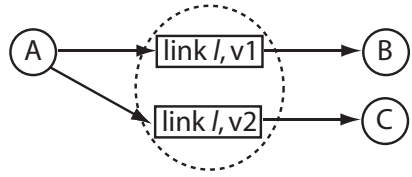


Fig. 1. First issue with link versioning

The first one that Østerbye mentioned in his paper is the navigation problem if a hyperlink has several versions. For example, in Figure 1, the version *v1* of a link connects two nodes *A* and *B*, and the version *v2* of that link connects *A* and *C*. The question is that when users navigate through the link, which destination

should be used. Human intervention would create cognitive overhead for users and would make the user interfaces for versioned hypermedia complex.

Another problem with the version control for links is shown in Figure 2. Suppose that we have "$A \rightarrow$ the link $\rightarrow B$". If B now is modified into B' (a new version *v2* of B), a new version of the link l must also be created connecting A to B'. The reason for this phenomenon is that links and resources have their own version histories. This is due to the fact most of existing versioned hypermedia systems followed either composition versioning or total versioning models.

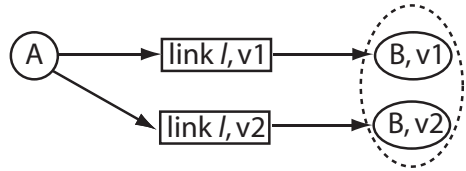


Fig. 2. Second issue with link versioning

The third problem with link versioning is whether a system should treat a hyperlink as a first-class object or a relation of anchor values. If the former one is chosen, the question is how the system determines when a new version of a link must be created [2]. If the latter choice is made, it is very hard for a system to distinguish two versions of a link that contains the exact set of member anchors or member resources.

3.3 Version control for hypermedia structure

The third structural issue that Østerbye mentioned is with version control for hypermedia structure. Hypermedia structure often refers to the network consisting of document nodes, anchors, and connecting hyperlinks. Østerbye suggested that the hypertext versioning system must allow users to return to a consistent previous state of entire network, rather than just a previous state of a single document node or link. Providing version control for both hypermedia structure and individual nodes/links is not easy to achieve [2].

3.4 Hypermedia element selection

A serious cognitive issue as providing versioning capability for hypermedia systems is associated with hypermedia element version selection. A prominent problem, when introducing versions of nodes, is to determine which element in the versioned group (i.e. a group of all object versions) the link points to [2]. In some systems (e.g. HAM, HyperPro), a link can point to either a specific element (i.e. a specific object version) or entire group. The link could also point to the current element, meaning the newest element in the versioned group. In a situation where the elements in the versioned group are organized in a version tree, it

is not clear which is the newest element. Some systems were based on a query language to do element selection. This increases the cognitive overhead for users, who have to specify a selection criteria each time a link is created. If a selection query must be specified each time a link is created, the users are discouraged from making links. Special care must therefore be taken at the interface level to simplify the use of link selection.

Another related issue is the place to store element version selection rules. There are two prominent choices. The first one is that element selection rules are stored on the structure, affecting all link endpoints, and provides selection of specific document revision from a versioned document (e.g. in HyperPro and HyperProp). With this approach, it is inefficient to evaluate rules across the revisions of a specific link [18]. The other popular choice is that the rules are stored on the containment arc between a container and its containees, providing selection of link revisions and document revisions from versioned links and versioned documents respectively. This structure versioning increases the work that must be performed to ensure that a structure container holds a consistent hypertext [18]. CoVer's [14] and VerSE's [15] followed this scheme.

3.5 Version creation

According to Østerbye, cognitive overhead can also increase with the version creation operation. If a user is forced to come up with names for each and every document node that is created, it will distract his/her attention from the actual subject. Similarly, if the user must explicitly create new versions of nodes or links all the time, and maintain some consistency among their versions, it will further distract his/her attention from the real work to be done [2].

4 Versioned hypermedia in Molhado

This section describes the Molhado versioned hypermedia model. More details can be found in another document [3].

4.1 Data model

The data model used in Molhado is the Fluid Internal Representation (Fluid IR). The main concepts in this data model are *node*, *slot*, *attribute*, and *sequence*. A node is the basic unit of *identity* and is used to represent any abstraction. A slot is a location that can store a value in any data type, possibly a reference to a node. A slot can exist in isolation but typically slots are attached to nodes, using an attribute. An attribute is a mapping from nodes to slots. An attribute may have particular slots for some nodes and map all other nodes to a default slot. The data model can thus be regarded as an attribute table whose rows correspond to nodes and columns correspond to attributes. The cells of the attribute table are slots (see Figure 3). Slots in an attribute table can belong to three types: constant, simple, or versioned slots.

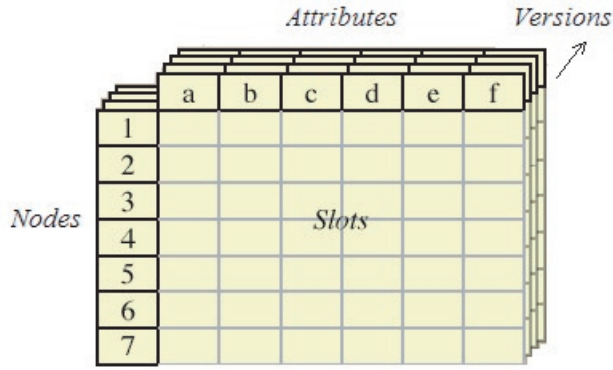


Fig. 3. Data model

A *constant slot* is immutable; such a slot can only be given a value once, when it is defined. A *simple slot* may be assigned even after it has been defined. The third kind of slot is the *versioned slot*, which may have different values in different versions (*slot revisions*). A *sequence* is a container with slots of the same data type. It has a unique identifier. Sequences may be fixed or variable in size and share common slots together. Once we add versioning, the attribute table gets a third dimension: the version.

4.2 Product versioning

The version dimension follows the product versioning model. Instead of focusing on individual components, Molhado versions a software project as a whole (see Figure 4). All system objects including (atomic and composite) components and hypertext structures are versioned in a *uniform, global version space*. Object properties can be defined as versioned or un-versioned. A *version* is global across the whole project and is a point in a *tree-structured discrete time* abstraction, rather than being a *particular state* of an object as in total and composition versioning models. That is, the third dimension in the attribute table in Figure 3 is tree-structured and versions move discretely from one point to another.

The state of the whole software system is captured at certain discrete time points and only these captured versions can be retrieved in later sessions (for example, the versions *v1.0*, *v2.0*, *v3.0*, etc in Figure 4). The *current version* is the version designating the current state of the project. When the current version is set to a captured version, the state of the whole project is set back to that version (for example, *v2.0* in Figure 4). Changes made to versioned objects of the project at the current version create a temporary version, *branching off* the current version. That temporary version will only be recorded if a user explicitly requests that it be captured. To record the history of an individual object, the whole project is captured. Capturing the whole project is quite efficient because the versioning system only records changes and works at a very small granularity [3].

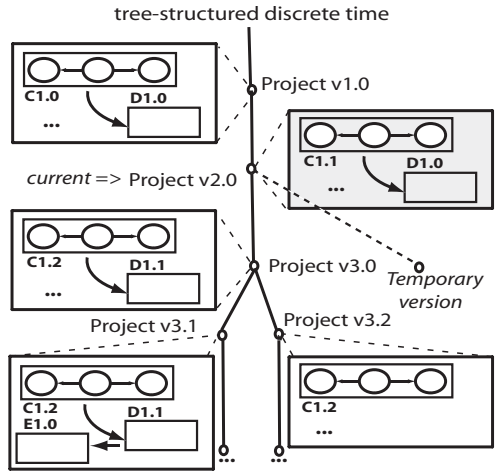


Fig. 4. Product versioning in Molhado

4.3 Structure-oriented representation

To be able to control versions of software documents at a fine granularity, Molhado follows a *structure-oriented* approach where each document is considered to be logically structured into fine units, called *structural units* or *logical units*. This approach is often taken in *structured document* research, e.g. SGML and XML. In this approach, each software document is represented by a *document tree* or a *document graph* in which each node encodes a logical unit of the document. Since XML has become the standard structured document format and very successful in representing many different data types, it is very natural to use XML for representing non-program artifacts. Syntactical rules for a document and its structural units are defined by users in a specification such as a *Document Type Definition* (DTD) or *XSchema* specification. For a program, an abstract syntax tree (AST) perfectly represents its logical structure.

To represent documents and hypermedia structure in this structure-oriented approach, tree and directed graph data structures are built from the nodes, slots, and attributes in the data model. A directed graph is defined with an attribute (the "children" attribute) that maps each node to a sequence holding its children. Trees additionally have the "parent" attribute, which maps each node to its parent. We have developed a fine-grained versioning scheme for tree and directed graph data structures in which fine-grained changes to the structure of a tree or a graph and to the contents of associated slots are recorded [3].

Figure 5 shows an example of our representation for an XML document. Each node in the XML document tree has an additional slot that stores its *operator* via "operator" attribute. An operator, which can be shared among nodes, identifies the syntactical type of its node and determines the number and syntactical types of the node's children. Nodes in an XML tree have operators drawn from two categories: *intermediate* and *text* operators. The unique *text operator* is used to

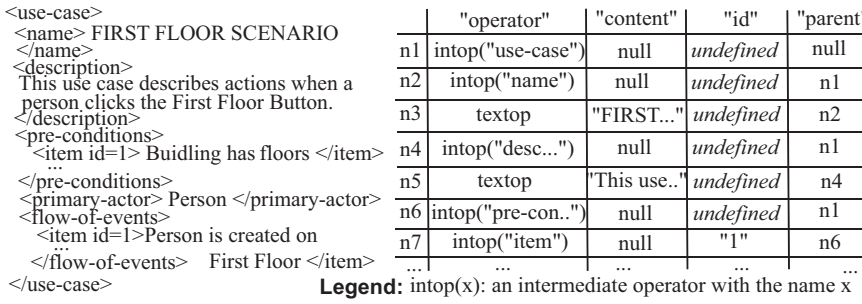


Fig. 5. Structure-oriented representation

represent XML's character data (CDATA) construct. Each node associated with the text operator has an additional slot (defined by the "content" attribute) that holds the CDATA string. Each element node in an XML document is associated with an *intermediate operator*, whose name is the element's name. Each node associated with an intermediate operator has one additional slot for each XML element-level attribute that is defined for that element (e.g. the "id" attribute). This document tree is versioned according to our tree-based versioning scheme.

4.4 Hypermedia entities

To enable HTML-style hyperlinks among these documents, an "href" attribute is defined for each node in a document's tree or graph. A "href" attribute contains a URL referring to a document node. Therefore, embedded hyperlinks can be attached to and can point to any document node.

The Molhado versioned hypermedia model is based on the following concepts: *linkbase*, *hypertext network*, *link*, and *anchor*. A linkbase is a container for hypertext networks and/or other linkbases. The relation between a linkbase and a hypertext network is the same as the relation between a directory and a file in a file system. A hypertext network can belong to only one linkbase. A hypertext network contains links and anchors. A link is a first-class entity and is an association among a set of anchors. An anchor can belong to multiple links. A link or an anchor can also belong to multiple hypertext networks. An anchor is used to denote the region of interest within a document, and it refers to either a document or a document node. This separation between anchors and document nodes allows for the separation between hypertext networks and document contents. Links and anchors can be associated with any attribute-value pairs.

4.5 Version control for hypermedia structure

To handle compositional relations among components or artifacts, Molhado also provides a structure versioning mechanism at both coarse-grained and fine-grained levels [3]. At the coarse-grained level, a composite component can contain atomic components and/or other composite components. At the fine-grained

level, an atomic component is allowed to have internal structure that can contain logical units.

A linkbase is implemented as a composite component, whose internal structure is a tree structure composing of other linkbases and/or hypertext networks. A linkbase is versioned according to the fine-grained versioning scheme for the tree data structure [3]. A hypertext network is implemented as a type of atomic component, whose internal structure is a directed graph. Each link or anchor is represented by a node in that graph. A directed edge connects an anchor's node to a link's node if the link contains the anchor. An additional attribute, attribute "ref", is defined for each anchor's node in the graph. The value of a "ref" slot is a *reference* to either a component or a logical unit within a component (but *not* to a hypertext network or to a linkbase). In general, via our versioning scheme for directed graphs, the history of a hypertext network is recorded [3]:

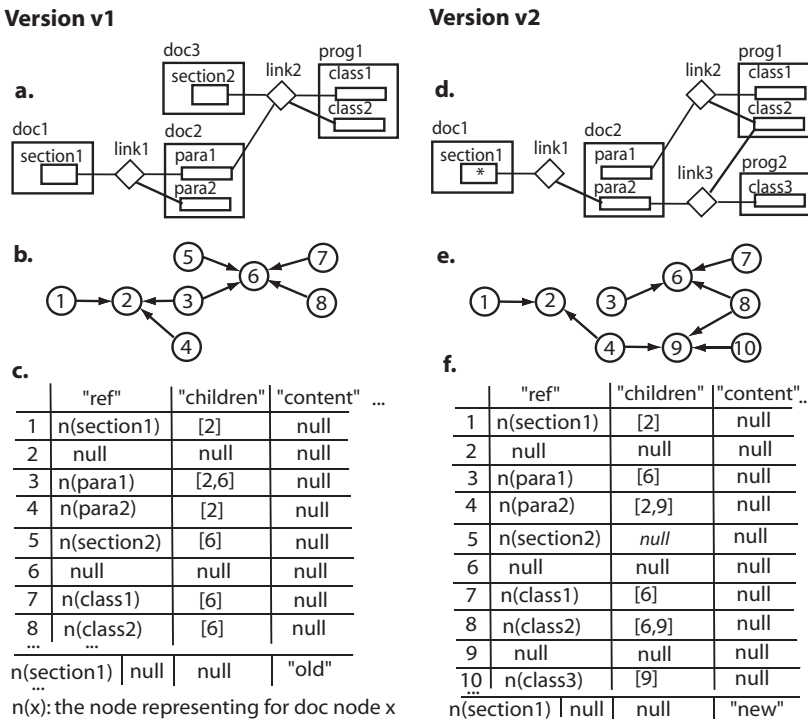


Fig. 6. Versioning for hypertext structure

Figure 6 shows an example of hypertext network versioning. Figure 6a) and d) display the network at two versions *v1* and *v2*. The directed graphs representing for structures of the network at these two versions are in Figure 6b) and e). Links' nodes (e.g. nodes 2 and 6) have edges coming into them and do not refer to anything. The attribute table is updated to reflect the changes to the hypertext

network. Figure 6c) shows part of the attribute table for the network at version *v1* (document nodes in the attribute table are not shown except n(section1)). In this example, attribute "content" defines for each document node a slot that contains a string value. The "ref" cell for an anchor node (e.g. node 1) contains a reference to the corresponding document node (e.g. n(section1)). Figure 6f) shows the attribute table at version *v2*. Node 5 was deleted since *doc3* was removed. Node 3 now has only one child. Node 9 (representing *link3*) and node 10 (representing *class3*) are just created. The "ref" cell for node 10 points to document node representing *class3*. The "content" slot of n(section1) has changed from "old" to "new" to reflect the change in the content of (section 1, doc 1) in the network.

In Molhado's product versioning model, a version is a point in tree-structured discrete time line. Therefore, returning to previous state of entire project is a basic versioning operation. When users set the current version to a recorded version, the state of entire software project will be set back to that version. Thus, documents as well as linkbases and hypertext networks will regain their contents and structures at that version.

4.6 Version control for links

Via the example in Figure 6, we can see that the directed graph versioning scheme also takes care of recording the history of individual hyperlinks (represented as a graph node). This is because the connection structure of a graph node and associated slots are captured over time.

Let us revisit the first issue with version control for hyperlinks (see Figure 1). In this case, Molhado avoids this problem. The nature of product versioning in Molhado allows developers to navigate to the right destination at the current version. In Molhado (see Figure 7), two global versions *v1* and *v2* would be created. At the version *v1*, the connection would be "$A \rightarrow$ the link $\rightarrow B$", and at the version *v2*, the connection would be "$A \rightarrow$ the link $\rightarrow C$". When the current version is selected explicitly or implicitly, the destination of the link will be automatically chosen at the current version.

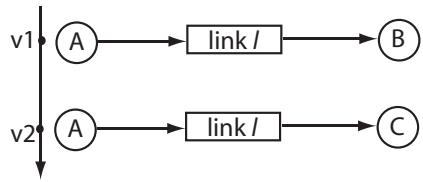


Fig. 7. No problem with link versioning in Molhado

Let us consider the second issue with versioning for hyperlinks (see Figure 2). This scenario can be handled by putting all hypermedia entities including links and resources in one global product versioning space such as in Molhado. Thus, when users select *v2* the current version, the destination of the link *l* would

be the new version of the resource B (see Figure 8). The key idea is that, in Molhado, link traversal occurs only among nodes at the *current* version.

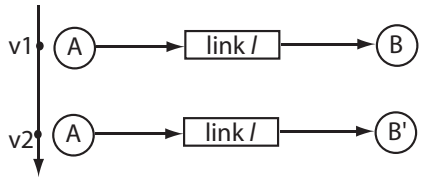


Fig. 8. No problem with link versioning in Molhado (2)

The third issue with link versioning is whether a system should treat a hyperlink as a first-class object or a relation of anchor values. In Molhado, a hyperlink is a first-class entity. It is represented as a node in a directed graph (representing a hypertext network). Each graph node might be associated with multiple attributes (representing properties of hypermedia entities). As explained, via the fine-grained versioning scheme for attributed directed graphs, a hypertext network as well as an individual link is versioned (see the example in Figure 6). In general, when a versioned attribute value associated with the hyperlink changes, or the connection of member anchors changes, or a document node on which a member anchor rests changes, a new version must be created. In this case, a new global version point is created in the tree-structured discrete time line, rather than a new version of resources or a new version of links.

In brief, Molhado provides configuration management for hypermedia structure as well as version control for individual links and document nodes without having problems that occurred in existing versioned hypermedia systems. Now, let us revisit other issues mentioned by Østerbye and relate them to Molhado.

4.7 Revisiting other issues

Immutability of versions: There are two characteristics in Molhado's product versioning model that suggested us a simple solution for this issue. The first one is that Molhado can control which aspects of a software project are versioned or un-versioned. The other one is that resource versions and hyperlink structure versions are maintained in the same global version space. To address this issue, in Molhado, users are allowed to specify the set of "substantial" links (i.e. the ones that if modified would create a new version), while all other links would be considered as annotation or comment links and would not require a new version if changed. Substantial links are modeled as *versioned slots* defined by "href" attribute, while other links as *simple slots* defined by an additional attribute. Therefore, when a substantial link is created and linked to a resource at the current version, a temporary version will be created and this change is hold at this temporary version. But when a non-substantial link is created, the change is hold at the current version and no temporary version will be created.

For hypertext networks, when a first-class hyperlink and its anchors are created and pointed to document nodes at the current version, a new version is created. This choice is reasonable since a first-class hyperlink and its anchors are considered as *annotations* on top of document nodes and are not parts of the document contents. In this case, the change between these two versions (the old and new ones) is the addition of the new first-class hyperlink and anchors. But the components themselves are not changed between these two versions.

Hypermedia element selection: With product versioning in Molhado, versioning for individual entities is subsidiary to versioning for entire project. That is, an individual entity do not have its own version numbers. The current version of entire project is selected implicitly (via user operations such as mouse actions) or explicitly (via user commands). Molhado knows how to relate objects and hypermedia structures at the current version together. Therefore, there is no version selection rules or selection queries involving in user operations on links, anchors, hypertext networks, and linkbases. For example, users' traversal can be done via HTML-style hyperlinks or via a hypertext network without involving users' selection of revisions of individual hypertext entities since the traversal occurs among nodes at the current version. When creating a link, users also do not have to make any selection about versions of targets. This characteristic of Molhado has facilitated the construction of a graphical user interfaces (GUI) for its versioned hypermedia services in the Software Concordance environment [31], which will be described later.

Version creation: In Molhado, individual document nodes or links share the same version space. To keep their version histories, a user has to commit changes to entire project. However, it does not require the user to check in or check out document nodes or hypertext elements individually. The user modifies documents and hypertext structures. When the user is ready to record the changes, he/she can issue a commit command and a new version will be created. Intermediate versions during an editing session can be used for undoing tasks but may not be saved in the SCM repository. In the total versioning model, when a hypermedia entity gets a new version, all other connected hypermedia entities (links, nodes, etc) and its ancestor entities in any compositional hierarchy need to be checked in individually. Although Molhado does not have the type of version creation problem that Østerbye described, a single change to a *versioned* entity might potentially create a new version of entire project.

5 The versioned hypermedia tool

5.1 Library functions

The Molhado's versioned hypermedia infrastructure is implemented in terms of a set of library functions. Versioned hypermedia functionality can be divided into two groups: 1) linkbase and hypertext network services, 2) link and anchor services. The first group includes functions to create a linkbase or a hypertext network, to delete existing hypertext networks or linkbases, to re-structure linkbases and relocate networks among linkbases, to open an existing hypertext

network, select a hypertext network to be *active*, to import and export a hypertext network from and to XLink format at any version. Services for links include link creation, deletion, renaming, attribute's value viewing, and link history viewing. Services for anchors include deleting an anchor, adding an anchor into the active link, removing an anchor off some link, renaming an anchor, and displaying the structural unit that the anchor refers to. More details on this set of library functions can be found in another document [32].

5.2 User operations

This set of library functions can be used in any editing or development environment. Automatic link generation algorithms could be employed to create links and our infrastructure could be used to store versions of documents and links. Molhado's hypertext versioning services were integrated into the Software Concordance (SC) development environment [31] (see Figure 9). Configuration management tasks are handled by Molhado's configuration management infrastructure [33]. The rest of this section describes the distinguished versioned hypermedia functionality in the SC environment.

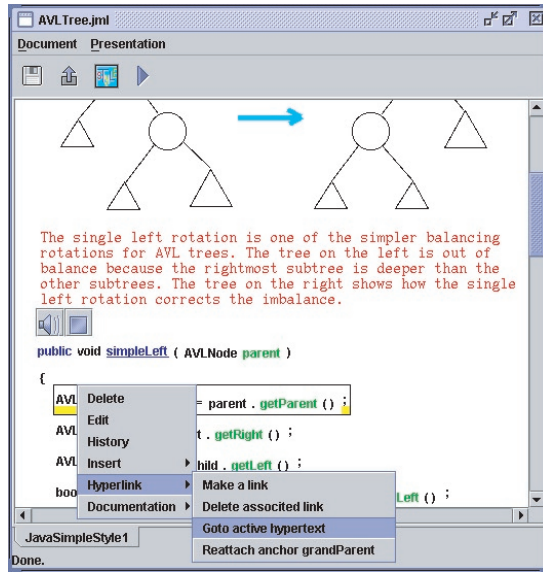


Fig. 9. Structured document editor

Molhado and the SC user interfaces support a variety of transactions. First of all, a user can open an existing project. After selecting the *current (working) version* from a project history window, the system displays the project's directory structure and documents in a project structure window. From this

window, the user can choose to edit, delete, import, or export any document and hypertext network. Also, via this window, the user can graphically modify the project's directory structure. If any modification is made to the versioned components at the current version, a new version would be temporarily created, branching off the current version. Subsequent modifications will not change the temporary status of the version until a *capture* or a *commit* command is issued. If the user does not want to keep the temporary version, he/she can *discard* it. Otherwise, the user can *capture* the state of the project at a version. The *capture* command changes a temporary version into a captured one. A unique name as well as date, authors, and descriptions can be attached to the newly captured version for later retrieval. A captured version plays the role of a checkpoint that the user can retrieve and refer to.

The user can *branch off* a version by just *switching* the current version and then starting the modifications. While working on one version, the user can always *switch* to work on (view or modify) any other version. This switching feature allows the user to work on many versions at the same time during one session. This capability is called *multi-version editing*. The user may *commit* changes at any time. Upon issuing this command, the user is asked which *uncaptured, temporary versions* should be saved and the chosen versions are then saved along with any already captured versions. Only the differences are stored. Saving complete version snapshots can improve version access time.

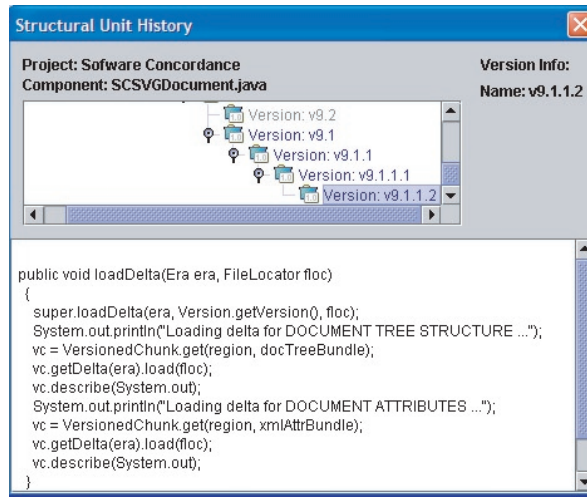


Fig. 10. History of a document node

The user can open a software document at the current version (e.g. XML, HTML, Java program, SVG graphic, UML diagram). An appropriate editor will be invoked. The editors are enhanced by versioned hypermedia services. Figure 9 shows the structured editor for a Java program. When the user right-clicks

on a document node, a popup menu is displayed to allow the user to create an anchor at the node and add it to the active hypertext network, to open the active hypertext network of an anchor defined at the document node, or to create or delete a HTML-style hyperlink at the document node.

In addition, the user is able to view a document node's history (see Figure 10). Note that the method "loadDelta" was not created until *v9.1*. Therefore, the earlier versions on the top window are "disabled". If the selected logical unit is the root of a document, the history of entire document will be displayed.

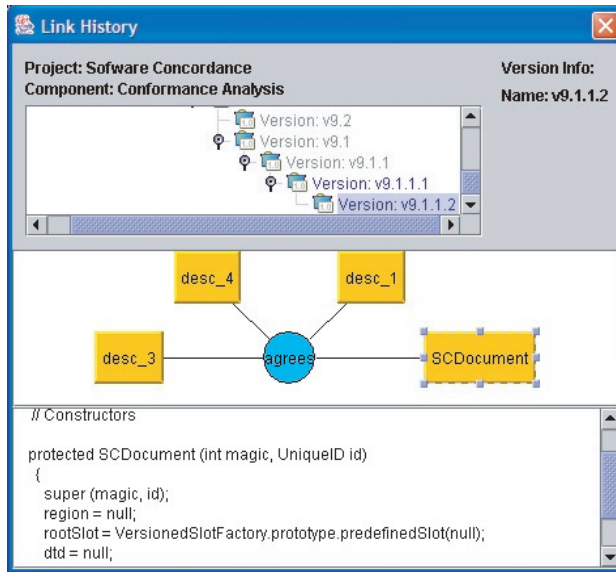


Fig. 11. History of a hyperlink

At any time, the user can create a hyperlink and make it active in a hypertext network. Then, he/she can add anchors into that hyperlink or any other hyperlinks currently managed in the system. Members of the hyperlink and its properties can be modified and changes are recorded. Figure 11 shows the history of the link "agrees" between several requirement/design items and the class "SCDocument". The constructor of the class was displayed in the bottom window since the user clicked on the corresponding anchor.

The user can also display, modify, or navigate through a hypertext network via a simple editor. In Figure 12, there are two types of links: causal and non-causal. There are directed edges coming from source anchors to a causal link, and from a causal link to its target anchors. There are only non-directed edges for non-causal links (e.g. link "n1"). While non-causal links are directly represented by our hyperlinks, causal links are extended from them with additional attributes. Via anchors, the user can traverse to corresponding document nodes.

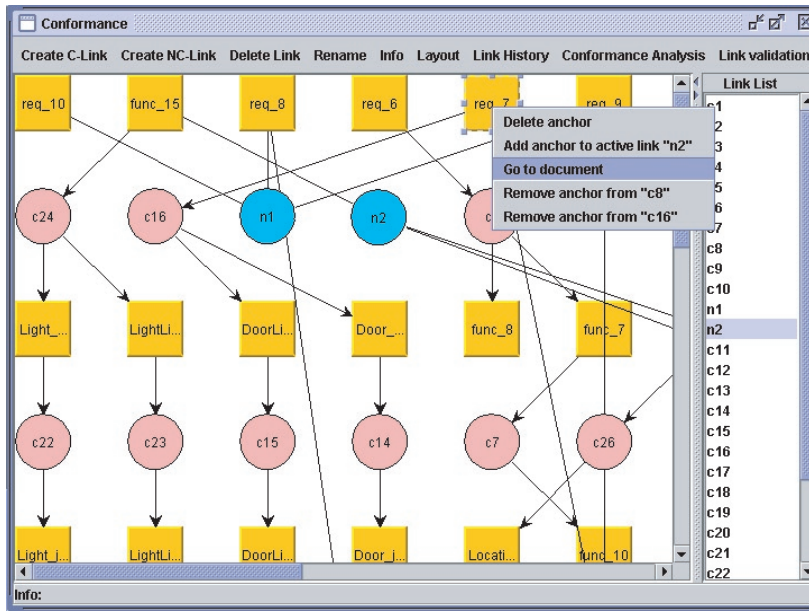


Fig. 12. A hypertext network

In this environment, multiple hypertext networks can also be defined during a software development process. With our tool, document contents will not be changed when a new network is defined since hypertext structure is separated from document content.

6 Issues for future research

While using product versioning, we have encountered three problems that have not been described for composition or total versioning systems. We plan to explore these issues in the near future.

6.1 Hyperlinks across different versions

Molhado's current interfaces are not sufficient to allow users to easily create hyperlinks between different versions. This is because links only exist within a single version. So, there is no way to say "make a link to document D at version v_k," because the linking interfaces have no parameter for the version. While this may not be the most common use case for versioned hypermedia, it is certainly necessary to support it and we have identified two approaches that should suffice.

The first solution is to allow links to other versions to be made as *computational links*. The anchors of a computational link are essentially queries against a database of links. Using the example mentioned above, the anchor's query would

be "get document D for version v_k." The second solution is to change Molhado's interfaces to include or even require a version identifier for all accesses. While this is logically possible, we have hesitated to support this choice because it is inconvenient in the most common cases.

6.2 Hyperlinks across different projects

Another similar issue concerns about how to create a hyperlink among components of different projects. Since a hyperlink is an entity that lives within a version space of a single project, with the current interfaces in Molhado, there is no way to do hyperlinking across different projects. However, the *computational link* approach should suffice to handle this. In this case, the query would ask for the opening of a different project, and loading a version and then a document.

6.3 The scope of the product versioning space

Molhado is built specifically for software engineering environments, where it makes sense for software developers to handle versions of entire software projects. Molhado's notion of a "project" is also applicable to Web-based applications or legal document management systems, representing an entire Web-based application, a Web site, or a collection of legal documents and tax codes.

However, we suspect that there will be hypertext collections that are ill-suited to being controlled in one version space. It is certainly the case that linking material maintained under Molhado's product versioning model with material managed using other versioning models will be a challenge. While this is clearly a topic for future research, we reject the idea that Molhado's apparent "incompatibility" with composition or total versioning systems is a reason to reject the product versioning model. Rather, we think that versioning needs a generalized access and maintenance model that covers all approaches.

7 Conclusions

The Molhado hypertext versioning infrastructure is well-suited for managing logical relationships among software documents. It is the first system that applies the product versioning model to hypertexts. The structural and cognitive issues that Østerbye [2] described has been well-addressed in Molhado by using product versioning. This version control model facilitates the development of a GUI for versioned hypermedia services. It reduces cognitive overhead for users in version creation and version selection of hypermedia elements in user operations. Software components and hypertext structures are uniformly versioned in a fine-grained manner. Users are allowed to return to a consistent previous state not only of a hypertext network but also of a single node and of a hyperlink. The use of product versioning for hypertexts also raises new issues that have not been described in existing versioned hypermedia systems such as creating hyperlinks across different versions or different projects. Our future plan includes exploring those issues and carrying out experimental and usability studies for the tool.

References

1. Whitehead, Jr., E.J.: An Analysis of the Hypertext Versioning Domain. PhD thesis, University of California – Irvine (2000)
2. Østerbye, K.: Structural and cognitive problems in providing version control for hypertext. In: Proceedings of the ACM conference on Hypertext and Hypermedia. (1992) 33–42
3. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C.: The Molhado Hypertext Versioning System. In: Proceedings of the Fifteenth Conference on Hypertext and Hypermedia, ACM Press (2004)
4. W3C: W3C XML Linking. http://www.w3c.org/XML/Linking (2005)
5. Nelson, T.H.: Xanalogical structure, needed now more than ever: parallel documents, deep links to content, deep versioning, and deep re-use. ACM Computing Surveys (CSUR) **31** (1999) 33
6. Akscyn, R.M., McCracken, D.L., Yoder, E.A.: KMS: a distributed hypermedia system for managing knowledge in organizations. Communications of the ACM **31** (1988) 820–835
7. Garg, P.K., Scacchi, W.: A hypertext system to manage software documents. IEEE Software **7** (1990) 90–98
8. Wiil, U.K., Leggett, J.J.: Hyperform: using extensibility to develop dynamic, open, and distributed hypertext systems. In: Proceedings of the ACM conference on Hypertext and Hypermedia, ACM Press (1992) 251–261
9. Ferrans, J.C., Hurst, D.W., Sennett, M.A., Covnot, B.M., Ji, W., Kajka, P., Ouyang, W.: HyperWeb: a framework for hypermedia-based environments. In: Proceedings of the Symposium on Software Development Environments, ACM Press (1992) 1–10
10. Cybulski, Reed: A Hypertext Based Software Engineering Environment. IEEE Software **9** (1992) 62–68
11. Tichy, W.F.: Design, implementation, and evaluation of a revision control system. In: Proceedings of the 6th International Conference on Software engineering, IEEE Computer Society Press (1982) 58–67
12. Asklund, U., Bendix, L., Christensen, H., Magnusson, B.: The unified extensional versioning model. In: Proceedings of the 9th Software Configuration Management Workshop, Springer (1999)
13. Delisle, N.M., Schwartz, M.D.: Contexts: partitioning concept for hypertext. ACM Trans. Inf. Syst. **5** (1987) 168–186
14. Haake, A.: CoVer: a contextual version server for hypertext applications. In: Proceedings of the ACM conference on Hypertext and Hypermedia, ACM Press (1992) 43–52
15. Haake, A., Hicks, D.: VerSE: towards hypertext versioning styles. In: Proceedings of the 7th ACM conference on Hypertext and Hypermedia, ACM Press (1996) 224–234
16. Delisle, Schwartz: Neptune: A hypertext system for CAD applications. In: Proceedings of ACM SIGMOD '86, ACM Press (1986) 132–142
17. Soares, L., Filho, G.S., Rodrigues, R., Muchaluat, D.: Versioning support in HyperProp system. Multimedia Tools and Applications **8** (1999) 325–339
18. Whitehead, Jr., E.J.: Design spaces for link and structure versioning. In: Proceedings of the conference on Hypertext and Hypermedia, ACM Press (2001) 195–204
19. Melly, Hall, W.: Version control in Microcosm. In: Proceedings of the Workshop on the Role of Version Control in CSCW. (1995)

20. Whitehead, Jr., E.J.: A proposal for versioning support for the Chimera system. In: Proceeedings of the Workshop on Versioning in Hypertext Systems, ACM Press (1994)
21. Hicks, D.L., Leggett, J.J., Nurnberg, P.J., Schnase, J.L.: A hypermedia version control framework. ACM Transactions on Information Systems (TOIS) **16** (1998) 127–160
22. Millard, D.E., Moreau, L., Davis, H.C., Reich, S.: FOHM: a fundamental open hypertext model for investigating interoperability between hypertext domains. In: Proceedings of the ACM Conference on Hypertext and Hypermedia, ACM Press (2000) 93–102
23. Wakeman, L., Lowett, J.: PCTE: the standard for open repositories. Prentice Hall (1993)
24. Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schutt, H., Thuring, M.: SEPIA: a cooperative hypermedia authoring environment. In: Proceedings of the ACM conference on Hypertext and Hypermedia, ACM Press (1992) 11–22
25. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys (CSUR) **30** (1998) 232–282
26. Lie, A., Conradi, R., Didriksen, T., Karlsson, E., Hallsteinsen, S., Holager, P.: Change oriented versioning. In: Proceedings of the 2nd European Conference on Software Engineering. (1989)
27. Cronk, R.: Tributaries and deltas. BYTE (1992) 177–186
28. Goldstein, Bobrow: A Layer Approach to Software Design. Interactive Programming Environments. McGraw-Hill (1984)
29. Reichenberger, C.: VOODOO: A Tool for Orthogonal Version Management. In: Proceedings of the Software Configuration Management Workshop, SCM-5, Springer (1995) 61–79
30. Wagner, T.A., Graham, S.L.: Incremental analysis of real programming languages. In: Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation, ACM Press (1997) 31–43
31. Nguyen, T.N., Munson, E.V.: The Software Concordance: A New Software Document Management Environment. In: Proceedings of the ACM Conference on Computer Documentation, ACM Press (2003)
32. Nguyen, T.N.: Object-oriented Software Configuration Management. PhD thesis, University of Wisconsin – Milwaukee (2005)
33. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C.: An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services. In: Proceedings of 27th International Conference on Software Engineering (ICSE 2005), ACM Press (2005)

Revision Control System Using Delta Script of Syntax Tree

Yasuhiro Hayase[1], Makoto Matsushita[1], and Katsuro Inoue[1]

Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{y-hayase, matusita, inoue}@osaka-u.ac.jp

Abstract. In an opensource development process developers work together using a revision control system. While getting multi-developers working products together into a single form, merge feature of revision control systems is used. Nowadays, merge operations in existing systems are commonly implemented with a line-by-line approach that can fail if two changes to the same line of code happen at the same time.
In this paper, we propose a two-way merge algorithm for source code that exploit the tree structure of modern programming language grammar: the source code is transformed in an intermediate XML representation and the merge operation is conducted on the transformed version.
We give an implementation of the algorithm for the Java language for the subversion revision control system.
Experiments shown that the proposed algorithm gives more accurate merge result than the existing line-by-line algorithms.

1 Introduction

The recent explosion in popularity of the Internet has attracted increasing attention on the open-source development process. In an opensource development effort, the developers distributed all over the world cooperate with each other in parallel, share information by email and mailing list, and manage the products with a centralized revision control system.

In such environment, each developer works side by side. And the results are stored independently in the repository of revision control system. At this time, the developer may have to merge his/her own modifications with the other developers' modifications. The merging operation can be done by humans, or done by the revision control system; usually the preference is given to the revision control system for the purposes of correctness and certainty.

The unit of merging of many existing revision control system is line of code. It may raise some problems when a merge is performed:

Illegal output Under the hypothesis that two developers are working at the same time on the same source file, if one developer deletes a variable and a statements that uses that variable, and the other developer adds a statement using the same variable, then the two modifications shouldn't be merged. However, "line of code" merging can't detect even if two changes conflict, and it outputs an illegal source code.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

Fail in feasible merging If one developer changes a statement and the other developer adds a comment to the line containing the previous statement, then these modifications can be merged. However, existing systems judge that these modifications cannot be merged, simply because they share the same line.

There is a method which recognizes syntax of the programming language and merges source codes more accurately as one of the methods of solving these problems. In [6] a syntax-based method for accurate merging is proposed. However, due the large number of programming languages it is hard to define an accurate algorithm for each language.

In this paper, we propose a merging algorithm independent from any specific programming language. The algorithm first builds an intermediate representation of the syntax tree of the merging source code before actual merging.

We use an intermediate language based on XML. Matchings of nodes and the deltas between trees are obtained by applying delta calculation algorithm based on FMES algorithm [3]. The delta is the sequence of operations required to modify tree. Merging is accomplished applying the sequence of operations to the tree. The language dependent procedure is separated from the merging procedure.

We implemented this algorithm on the actual revision control system subversion [1]. The system targets Java source codes, and correctly deals with deletion of variable and moving of fragment of source code.

Moreover, we compared our merging algorithm with the algorithm implemented in other system. Concretely, we compared the output of our system and existing system using sample source codes and source codes extracted from development history of actual opensource development. As a result, our system merge achieve a more accurate merging than existing system.

The paper consist as follows: Section 2 represents common revision control system and shows problem of existing revision control system. Section 3 explain storing the intermediate tree representation of the source code in repository. Section 4 shows merging system which recognizes syntax of source code as a solution of the problem. Section 5 explains the implementation of the system. Section 6 shows experimentation and consideration. Section 7 compares our system and related works. Finally concluding remarks and future work are given in Section 8.

2 Revision Control System

First, we represent revision control system used in opensource software development commonly. And next, we explain by using the example of the problems when the results of the works concurrently done are merged.

2.1 Software Development using Revision Control System

Revision control system (e.g. CVS) manages modification history of files such as source codes and documents.

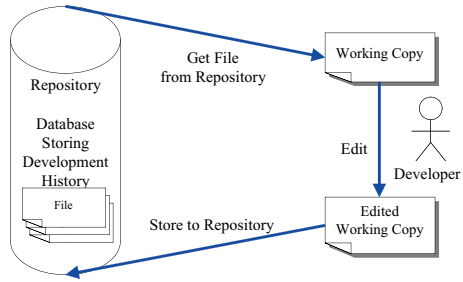


Fig. 1. Revision Control System

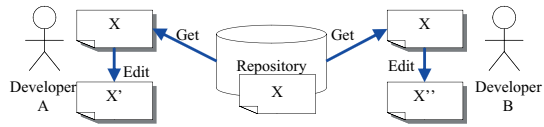


Fig. 2. Parallel development by multi developers

Figure 1 shows a simple workflow of software development using revision control system. Revision control system has a database called **repository** which stores all of developed files. Developer must get copy of a file called **working copy** from repository before modifying the file. And after the developer finish modifying, the developer stores the file in the repository. Software development using revision control system is achieved repeating this workflow.

Also, many of revision control systems support the software development by multiple developers. Figure 2 shows that developer can get copy of a file even if other developers hold copy of the file, and that each developer can modify working copy at will.

When developers modify a file in parallel in this way, modification except for last stored one is lost if each developer stores result of own modification only. To prevent losing other developers' modifications, the file is needed which includes own and other developers' modifications (Figure 3). The procedure getting such file is called **merging**. Merging is done by revision control system or by hand.

2.2 Problems on merging

Existing revision control systems merge line-by-line, and detect conflict if and only if the same line is modified. It has problem of detecting unnecessary collisions or outputs illegal source codes. Following sections show each problem and ideal solution.

Unnecessary collision Suppose that developers A and B get working copies of same file and modifying it. The file had following line.

```
int refs;
```

Developer A modified the line to initialize variable `refs` as follows and stored

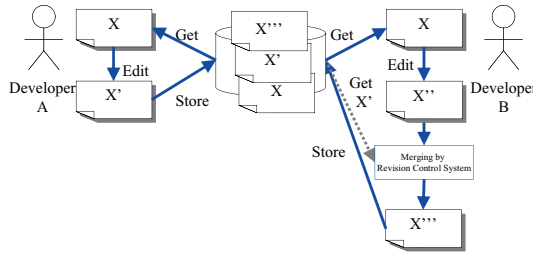


Fig. 3. Commit merged result

to repository.

```
int refs=0;
```

Developer B added comment about variable `refs` in same line as follows before knowing the modification of A and attempted to store.

```
int refs; /* reference count */
```

Developers A and B modified same line, so the revision control system judge that these modifications conflict. In this case, developer B must resolve the conflict because B stores the file after A.

If the system recognizes modifications right way, the system should output following code having both initializing and comment.

```
int refs=0; /* reference count */
```

Illegal merging output Please assume developers A and B get working copies of same file and modifying it. The file had following line which declares three variables.

```
int num, sum, avg;
```

Developer A considered that variable `avg` is not necessary, and deleted it as follows.

```
int num, sum;
```

Developer B added a statement using variable `avg` as follows before knowing the modification of A and attempted to store.

```
int num, sum, avg;
⋮
avg = num/sum;
```

Developer B have to merge own modification and Developer A's modification before storing. In this case, the revision control system outputs source code as follows because the modifications are not on same lines.

```
int num, sum;
⋮
avg = num/sum;
```

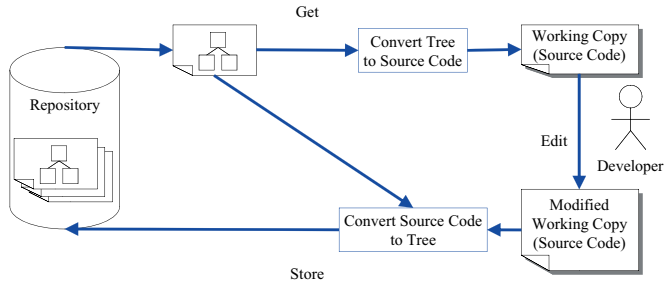


Fig. 4. Data flow on storing modified working copy

However, undeclared variable `avg` is used in this source code. This result contradicts the intention of B. If a variable having same name is declared at outer scope, the developer might not sense the problem because no error is detected on compiling this source code.

If the system recognizes modifications right way, the system should detect collision.

3 Storing source code to repository

In order to resolve problems described in Section 2.2, we propose a revision control system which stores in the repository not the source codes but the intermediate tree representation of the source code. The tree is a syntax tree with white space strings and comments. Specifically, the tree is an ordered tree with no limitations in the number of child nodes. All nodes of the tree have unique IDs. Details about IDs are described in Section 3.2. Data structure of the tree is independent from programming language, so the system does not depend on specific programming language.

3.1 Analyze source code to syntax tree

When developer adds a new source code, the system automatically build a new tree by parsing the source code, and add new unique IDs to all nodes of the tree. When a file is checked out, the system fetch the matching tree from the repository and converts it to source code (Figure 4). The source code is obtained from the tree by doing a depth-priority-search visit and printing the text stored in the node. Developers can then modify the source code and commit the changes to the repository. When the system stores the source code, it parses the source code again, makes a tree without IDs. And it searches for any pairs of similar nodes between the modified source tree and the original source code tree. Any node in the modified source code tree that belongs to the pair has assigned the same ID of the other node in the pair. Any node that doesn't belong to some pair has assigned and unique ID by the system.

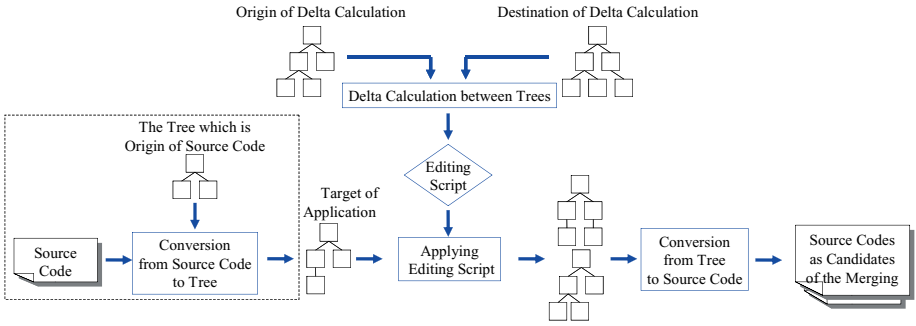


Fig. 5. Data flow on merging

3.2 Adding IDs to nodes

To identify the same node in different trees, the system put IDs to nodes. The procedure to put IDs to the nodes is as follows. The tree that is the origin of the source code is stored in the repository of the system, and the nodes of the original tree have IDs already. First, the system searches for the pairs of two nodes by comparing A and B. This procedure is called **matching calculation** at the following. The result of the calculation is a bijective map f between nodes in A' and B' which are the subsets of nodes in A and B respectively. The map f associates nodes which are judged identical on the basis of the data in the nodes or similarity of parent or brothers. Nodes in A' get the same IDs as nodes which correspond in f respectively. Nodes not in A' but in A get new unique IDs respectively. The FMES algorithm[3] is used for matching calculation. Then, the system makes links from the nodes which use a variable or a function to the nodes which declare them.

4 Merging algorithm recognizing syntax of source code

In this section, we explain the merging algorithm that exploits the syntax of the source code. This algorithm targets the tree described in section 3.

Figure 5 shows data flow in merging procedure. The inputs of the procedure are (1) the modified source code tree, (2) the original source code tree, and (3)the destination source code tree.

At first, the difference between (2) and (3) tree are computed. The difference is expressed as a sequence of tree editing operations called **editing script**. Multiple trees are output by applying the editing script to the first tree.

4.1 Delta calculation between trees

The delta between trees is expressed as an **editing script**. An editing script is a sequence of **tree editing operations** of four kinds:

1. `insert`: Add a node to the tree. Its parameters are the ID of the node, the data in the node, ID of the parent node, and a number which represent position in the brothers.
2. `delete`: Delete a node from the tree. Its parameter is the ID of the node.
3. `update`: Update the data of the node. Its parameters are the ID of the node, and the new data in the node.
4. `move`: Move a subtree around the tree. Its parameters are the ID of root node of the subtree, ID of the parent node in destination, and a number which represent position in the brothers.

The system edits a tree by applying sequentially the operations described in the editing script.

Generally, for two arbitrary trees A and B, there are an infinite number of scripts that may convert A into B. We define the cost of the editing script as the summed cost of each single operation in the script, and we adopt the script with lowest cost. The cost of operation is defined in delta calculation algorithm. We use FMES algorithm [3] extended by analysis of renaming as follows.

Renaming analysis FMES is a delta calculation algorithm for generic tree based on text similarity. Some problems can arise if it is applied unchanged to source code. There is a problem on identifiers matching. For example, it is typical that variable having similar role are named in a similar way, for example a common prefix and number or short name suffix. (ex. `buffer1, buffer2...`) These names are similar enough as text, so FMES may judge `buffer1` and `buffer2` are matched. When the variable's name is changed, text-based matching algorithm can't match renamed nodes.

Thus, a different algorithm is used for matching of nodes representing identifiers. First, matching of all the nodes but the identifiers' ones is calculated with the FMES algorithm. Next, the algorithm makes two sequences of the identifier nodes of each tree, and calculates the longest common subsequence (LCS) of nodes. The identifier nodes match if the nodes have exactly the same data or the ancestor nodes of the identifier nodes match at more than the certain rate.

The pairs of nodes in the LCS are added to the matching set. Matchings of the other identifier nodes are calculated by brute force method.

4.2 Applying editing script

This section describes the algorithm which applying the editing script from tree A to tree B to yet another tree C. The inputs of the algorithm are the editing script S, the tree C as target of editing, and the tree A as the origin of S.

The editing operations in the script use the node's ID to identify nodes. However, when applying S to C, for a given ID, there may be no matching node in C. In this case, similar nodes substitute the target node of the operation. (Figure 6)

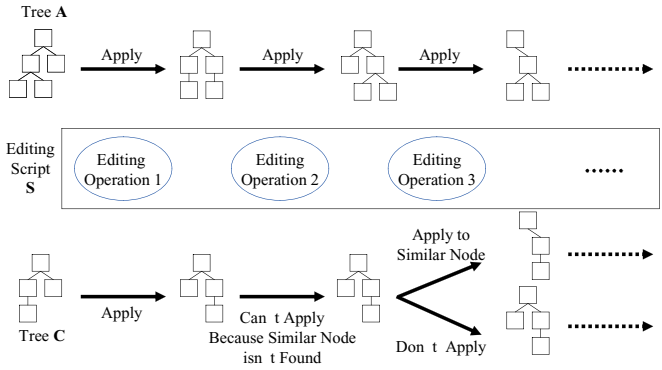


Fig. 6. Applying editing script to source code tree

Searching similar nodes and positions Similar nodes are searched in two ways. The first method is based on the similarity of sibling nodes (figure 7(a)). Consider the set made of all the pairs of nodes on both sides of the target node, and the subset of it whose elements satisfy the following conditions:

- Two nodes which have the IDs of both of pair exist in tree C.
- The two nodes in tree C have same parent node.

Only the pairs with the shortest distance between its nodes are picked up from the subset. The candidate nodes are searched between nodes of tree C corresponding to both of the pair. The candidate nodes must have similar text data, and the node having same ID as the candidate node must not exist in tree A.

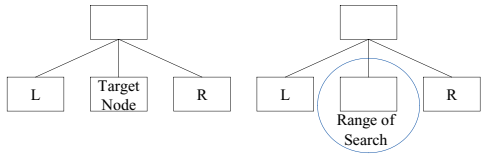
If target node exists on edge like figure 7(b), then the candidate nodes are searched using brother nodes in one side as a clue.

The second method uses parent node. Like figure 7(c), the candidate nodes are searched from children of the node having same ID as the parent node of the target node. The candidate nodes must have similar text data, and the node having same ID as the candidate node must not exist in tree A. This method is used only when the ID of parent of target node was not used in the first method.

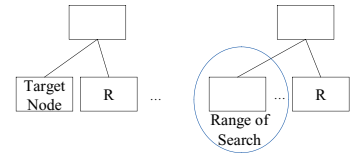
The candidate nodes founded by these methods have a score. If a candidate node has same ID as target node, then the candidate node has 3 point. Else, the candidate node gets 1 point if each of left, right, or parent of the candidate node has same ID as corresponding node of target node.

The position as parameter of editing operation described as the ID of the parent node and the rank order in the brothers. Accordingly, every time when an editing operation is applied to tree C, it is necessary to search the similar position. The candidate positions are searched and scored based on left, right, and parent node of the target position as well as the candidate nodes.

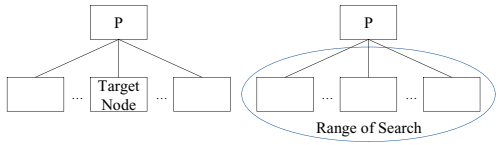
Applying editing script to original tree It is necessary to apply the same part of the script S to the tree A in order to search similar nodes and positions. Thus the script S is applied to tree C and tree A simultaneously.



(a) from brother nodes



(b) from brother nodes (when target node exists on edge)



(c) from parent node

Fig. 7. Finding similar nodes

Applying editing script When applying the edit operations, if multiple candidate nodes or positions are found, then a tree is built for each possible combination of nodes.

When applying a `delete` operation whose target node has child nodes, two trees are built: one with the removed subtree, and one copy of the original tree.

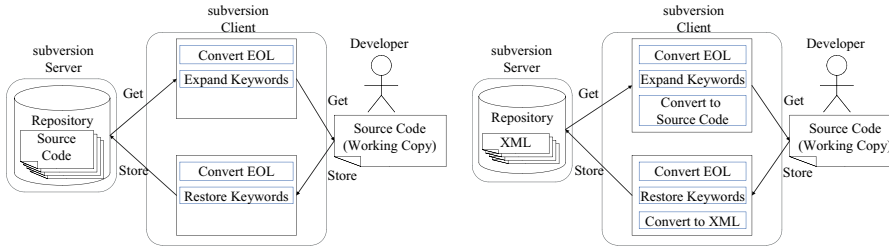
The algorithm records score of trees. Before applying the script, the score of tree is 0. When the algorithm applies each operation, The value calculated from how operation was applied is added to the score. The value is a score of the node or the position to which the operation is actually applied. In the case of `move` operation, the value is average of scores of the node and the position.

5 The implementation

This section describes the implementation of the system. We implemented the system by extending the Subversion revision control system. The implementation of the checkout and commit operation is described in section 5.1, and the implementation of the merging operation in Section 5.2.

5.1 Checkout and commit

Figure 8(a) shows the original checkout and commit operations as implemented in Subversion. Subversion is a client-server system, whose server manages the repository and the client manages the working copy. The client hash function converts the EOF character and replaces certain keywords when creating a working copy by checkout or committing to the repository.



(a) Checkout and commit of original subversion

(b) Checkout and commit recognizing tree structure of source code

Fig. 8. Checkout and commit of existing system and our system

We added the following functions to subversion client.

1. Conversion from tree to source code
2. Conversion from source code to tree

The tree is expressed in special intermediate language based on XML. The subversion server has not been modified. Figure 8(b) shows the outline of the extended subversion system. If the file which checked out or committed has no special meta data `svn:conversion-handler`, extended subversion behaves same as original subversion.

If the file has meta data `svn:conversion-handler`, the XML file is stored in the repository. When the extended client checks out the file, the client converts the XML file to source code and uses the source code as working copy. When the extended client commits the file, the client converts the source code to an XML file.

The following sections describe the implementation of conversion between source code and XML file.

Conversion from source code to XML file The system and make syntax trees keeping white space strings and the comments by analyzing the source code. The system creates an XML file whose elements all correspond to all nodes of the syntax trees. The white space strings and the comments are also included in the XML file.

And the system makes links from an identifier element using a variable or a method to identifier declaring the variable or the method. The following methods are used to express the links. The system generates a temporary ID, and put it to attribute `id` of the destination element of the link, and put the same temporary ID to attribute `ref` of the source element of the link.

Next, the system put unique ID to all elements of the XML file. When new file is added to repository, the system put new ID to all elements of the XML file.

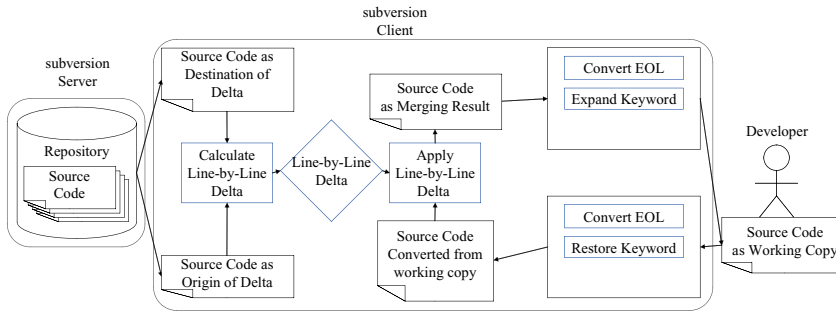


Fig. 9. Merging on original subversion

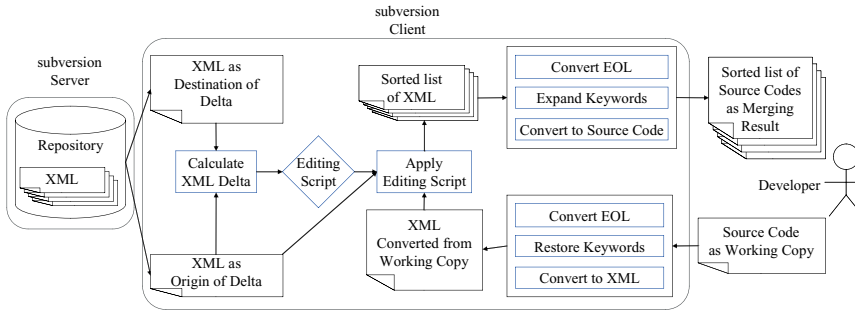


Fig. 10. Merging recognizing tree structure of source code

When file is not new one, the XML file before editing is stored in the repository, and elements of the XML file have already unique IDs.

Then, the system calculates the matching of nodes between the XML file to be committed and the XML file before editing. This calculation uses FMES algorithm and analysis of renaming described in Section 4.1. If the element in the new XML file corresponds to an element of old XML file, the system puts same ID as the element in old XML file to the element in the new XML file. If the element in the new XML file does not correspond to any node, the system puts new ID to the element.

The Universally Unique Identifier (UUID) is used as new ID. When the system puts ID to the element having temporary ID, the system also changes the ref attribute of elements which link to the element.

Conversion from XML file to source code The XML file is converted into the source code only by removing all tag from the XML file, because the XML document keeps all texts of the source code as a text element in order of appearing.

5.2 Merging

Figure 9 shows data flow when original subversion system merges files. The original subversion client merges files line-by-line.

The inputs of merging procedure are a working copy and the two files in repository. First, end-of-line character and keywords in working copy are restored. And, calculate the line-by-line delta of two files stored in the repository. Next, the client applies the difference to the restored working copy. Finally, the system expands the keyword and converts end-of-line characters of the application result. The working copy is overwritten using it.

We added to the subversion client the following functions.

1. Conversion from tree to source code
2. Conversion from source code to tree
3. Delta calculation between trees.
4. Delta application to tree.

Function (1) and (2) are described in section 5.1. Only the client is extended, and the server isn't changed. Figure 10 shows the outline of the merging function of our extended subversion. If the file has no meta data `svn:conversion-handler`, our system processes same as the existing subversion client.

If the file has meta data `svn:conversion-handler`, the system uses function of merging tree. In this case, as described in section 5.1, an XML file is stored in the repository and working copy is a source code.

First, the working copy edited by developer is converted into XML file as well as storing it. And, calculate the tree delta of two XML files stored in the repository. The delta is expressed as an editing script. The merge results are obtained by applying the delta to the XML file made from working copy.

The merging result has two cases. One is the single XML file is obtained. And another is the multiple XML files are obtained. If single XML file is obtained, the system converts the XML file to source code and overwrites the working copy using it. If multiple XML files are obtained, the XML files are sorted by the score described in Section 4.2. The developer must select one XML file. The system convert selected XML file to source code and overwrites the working copy using it.

Applying of delta When the system applies the delta to the tree, for the purpose of searching similar nodes or positions, the same delta is applied to tree which is the calculation origin of the delta at the same time. The system searches neither similar nodes nor the positions when the delta is applied to the tree which is the calculation origin of the delta.

Similar elements are searched when the element having target ID of operation does not exist or when the operation to the text element is applied. The `insert` operation and the `move` operation take a position in the parameters. So, similar positions must be searched when these operations are applied. The similar nodes and positions are searched by using the method described in Section 4.2.

When multiple similar nodes or positions are found or when any nodes or position are not similar enough, the tree is duplicated. The next operation in the editing script is applied to the duplicated trees as well as duplication origin.

The system verifies whether the trees to which all operations were applied fulfill the following requirements.

- The destination of link from identifier node does exist. If the destination doesn't exist, the system warns to developer.
- The names of the identifier are equal between the link source and link destination. If the names are different, the identifier has been renamed. The system changes the name of the link source to the name of the link destination.

6 Evaluation

In this section, we show the two experimentation results of our system.

6.1 Application to sample source codes

We checked whether our system can correctly merge the source codes made for the test.

At first, we made a source code. It is called the Original. And we made the three source codes that were made by modifying the Original in different way. Each source code is called the Variant 1, 2, 3. Each modification is described below.

Variant 1 The variable `x` declared in Original was deleted.
Variant 2 The method using variable `x` was added.
Variant 3 The variable `x` declared in Original was renamed.

The tree deltas from Original to Variant 1, 2, 3 were calculated. Each delta is called the Delta 1, 2, 3. We applied these deltas to the Variant 1, 2, 3 and check the result. Moreover, we did same experiment using line-by-line merging and compared the results. Table 1 and Table 2 show the experiment result.

The line-by-line merging system failed all combinations. It output an illegal source code or detected wrong conflict. (Table 1(a))

On the other hand, our system correctly merged excluding one. It output right source code or detected correct conflict. In one case that calculation failed, the calculation did not finish because it failed in the search for a similar position, and it had generated a huge amount of trees.

We recognized that we should improve the precision of the search for the similar nodes and the positions from this experiment result.

	Delta 1	**Delta 2**	**Delta 3**
Variant 1		Illegal Output	Conflict
Variant 2	Illegal Output		Illegal Output
Variant 3	Conflict	Illegal Output	

(a) Result of line-by-line merging

	Delta 1	**Delta 2**	**Delta 3**
Variant 1		Failure	Success
Variant 2	Dead link detection		Success
Variant 3	Success	Success	

(b) Result of tree merging

Table 1. Result of merging

6.2 Pseudo application to development of actual softwares

Target of the experiment We extracted the revision guessed to have used the merging function from the warehouse of actual opensource software. And we tried to reproduce the files before it was merged in those revisions. It is as follows concretely.

Please assume developers X and Y committed the same file F at short interval. Developer X committed at revision n and Developer Y committed at revision n+1. When the range of the change in revision n+1 is included within the range of the change in revision n, it is considered that developer Y checked out revision n, and edited it. When it is not so, it is considered that developer Y checked out revision n-1, edited it, merge the modification of developer X to working copy, and committed it.

The file before developer X change is merged is not stored in the repository. However, this file can be reproduced if it correctly merges it with the opposite direction. Then, we did the experiment that tried to reproduce this file using the merging function.

84 data in which committing happened within ten minutes was extracted from the repository of the Eclipse project (22606 files and 162683 revisions) and Jakarta projects (19420 files and 103358 revisions), and the data is used to experiment.

line-by-line merging	count	tree merging	count
Success	71	Success	71
Failed	13	Success	9
		Failed	4

Table 2. The result of merging actual softwares

cause of failure of line-by-line merging	count	tree merging	count
Addition or Deletion of White Space to the Same Line	4	Success	4
Semantic Change and Reform	1	Success	1
EOL Code Change	1	Success	1
Overlapped Semantic Change	2	Success	2
Overwriting Prior Change	2	Success	1
		Failure	1
Semantic Conflict	2	Failure	2
Broken Source Code	1	Failure	1

Table 3. details when line-by-line merging fails

Result of the Experiment Table 2 shows the result of the experiment. The tree merging has succeeded when succeeding in line-by-line merging. In addition, tree merging has succeeded in 9/13 in which line-by-line merging failed.

Next, details when line-by-line merging fails are shown in Table 3.

The reason for two cases of four cases in which "tree merging" failed is that it had tried to merge two edits conflicting semantically. A huge amount of candidates was generated when the editing operation was applied, and it failed in merging though edits are not conflict semantically.

7 Related Works

Wuu Yang [11] proposed the algorithm identifying syntactic differences between two versions programs.

T. Mens [8] classifies technologies of software merging by various criteria. Our algorithm is basically classified in three-way, syntactic, and operation-based merging.

Bernhard Westfechtel [10] proposed the syntactic 3-way merging algorithm. The algorithm is consist of language-dependent part and language-independent part, and detects conflict using the information of identifier. Our system is different from the algorithm of Westfechtel in two points. First, our system allows user to use arbitrary editors to edit the source code. Second, our system may output two or more candidates of merging result.

David Binkley et al. [2] proposed the semantic merging algorithm targeting the simple language having procedure call.

Marc Shapiro[9] and A. Kermarrec[7] propose the method of the synthesis of editing scripts by calculating the dependence between the operations in the each editing script, and re-ordering the operations in the scripts.

XmlDiff, DeltaXML, XML TreeDiff, diffmk, xydiff, etc. are enumerated as a system that calculates the difference of a general XML document. Because this research puts ID to the elements of the XML files, and simplifies the application of delta. This research distinguished from the above systems.

XCoP and Oracle XML DB are enumerated as a system that stores the XML document in the database and manages its version. These systems use the special operation for the editing of XML document. In our system, the developer can use arbitrary editor for editing source codes.

Shu-Yao Chien [5] [4] propose the method of making the time cost when an old version is acquired united to the amount of the disk use when the history XML document is stored.

8 Conclusion

This paper shows the problem of the merging function of existing revision control systems, and proposed the merging function according to the syntax of the source code as one solution to the problem, and described the design of the system. This system decreases the conflict on merging, and lightens the problem caused by merging.

The work in the future are covering the languages other than Java, improving precision of the matching, merging more efficiently, and spanning link between different files in the repository. It is also important to cover the document forms other than the source code.

Extension to Subversion done in this research is distributed as free software at following URL.
http://sel.ist.osaka-u.ac.jp/˜y-hayase/svn-xml/

References

1. subversion. http://subversion.tigris.org/.
2. D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.
3. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.
4. S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Version management of XML documents. *Lecture Notes in Computer Science*, 1997:184–189, 2001.
5. S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. XML document versioning. *SIGMOD Record*, 30(3):46–53, 2001.
6. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

7. A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.
8. T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.
9. M. Shapiro, A. Rowstron, and A.-M. Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), 2000.
10. B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, New York, NY, USA, 1991. ACM Press.
11. W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.