SOFTWARE CONFIGURATION MANAGEMENT: ORTHOGONAL DIMENSIONS FOR SYSTEMS EVOLUTION

Luis Lima
College of Management and Technology – ESTGF / Polytechnic Institute of Porto
Felgueiras, Portugal
llima@estgf.ipp.pt

Carlos Ramos
Institute of Engineering – ISEP / Polythecnic Institute of Porto
Porto, Portugal
csr@isep.ipp.pt

Paulo Novais
Department of Informatics – School of Engineering / University of Minho
Braga, Portugal
pjon@di.uminho.pt

Abstract – This paper presents an overview of the tasks and limitations of traditional Software Configuration Management systems. In addition to the familiar check-in/check-out model, a brief characterization of another three configuration management models found in SCM tools is presented. Three dimensions for software systems evolution are considered: time, space and team cooperation. A case study is presented to exemplify the three dimensions of evolution. Reasoning for the need of a new data model representation supporting the three dimensions, viewed as orthogonal to each other, is also presented.

Keywords: Software Configuration Management, software architecture, version control.

I. INTRODUCTION

In the process of development of a software product, the software engineer must be concerned not only with the development itself but also with the support to the product evolution. The support to the evolution is a consequence of the need to preserve several stages of the product, resulting from the existence of more than one instance of the (almost) same product at the same time or resulting from timed snapshots, probably obsolete, of the product.

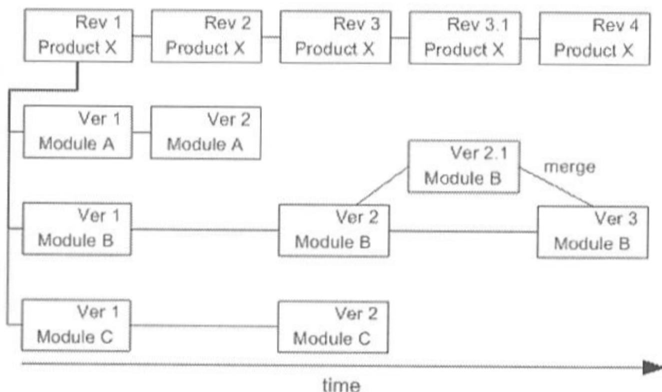

Figure 1- Evolution over time

Figure 1- shows a product as it evolves over time. The product is composed of three modules. The evolution is shown as two superimposing graphs: one represents the successive revisions of the software product and the other represents the versions of all the artifacts needed to build the product. Each node of the graph represents a new version/revision, approved and immutable. The transitions between nodes represent the transformations from one version to the next. A new revision of the software product is build whenever an artifact is modified. Some modifications are temporary, maybe to make urgent corrections (Module B – Ver 2.1), and are merged with the next major version of the artifact, as soon as possible.

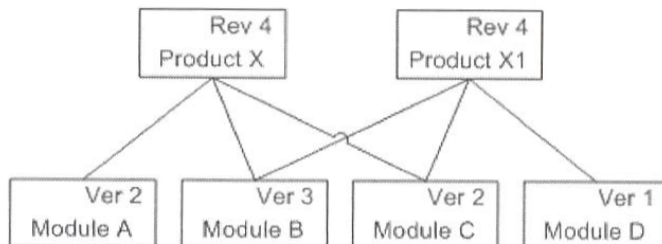

Figure 2- Permanent variant of Product X

Figure 2- shows a permanent variant of the software product. Product X1 is a variant obtained substituting module A by module D. Module D could be itself a permanent variant of module A. From this point on the two software products coexists and evolve separately. A new version of a common component results in two new revisions, one for each product variant. Product X1 exhibits the main features of product X but with some feature customized for a specific context or client. The problem becomes increasingly complex as more variants are created.

The record of the several stages of an evolving product is a pre-requisite to audit the product and to identify the transformations between consecutive stages. It also allows going back to a previous stage if the new one is not viable. The new stage may not be viable because of a late recognition of an incorrect evolution path. Sometimes, it is not viable because the transformations, eventually complex, from one stage to the next, introduced one or more errors that are more easily eliminated going back to the previous stage and making a different set of transformations.

The effort to quality control also requires an effective management of the process of the product development. The process must be auditable in order to enable the evaluation, measurement and control, only possible thru

the systematic record of the processes and intermediate products.

Also, the pressure to quickly release a new product demands for parallel development. The team effort to cooperatively develop a product usually brings along the need to share software artifacts, themselves still in development. The mechanisms for the control of shared use of those artifacts, with or without concurrent changes, are essential to this effort.

The remainder of the paper is organized as follows. Section II elaborates on traditional SCM tasks and tools, focusing on the different models embodied on those tools and their limitations. On section III an example of an evolving software product is outlined, illustrating the context for the use of SCM tools and techniques. Section IV presents a proposal for a data model formalization, capable of representing an evolving software system over three orthogonal axis. Finally, on section IV the conclusions are presented.

II. Software Configuration Management

These questions have been addressed by Software Configuration Management (SCM)[1]. Some problems are still yet to solve, mainly because the same mechanism is used to address very different situations, such as:

- Temporary versions for corrective maintenance;
- Variants of the same software product, modified only to work on different hardware platforms;
- Cooperative work, as needed to support team development;
- Control of historic revisions as the product evolves over time.

The artifacts handled by traditional SCM are text files, corresponding to source code of some programming language. So, it is not possible to dissociate the physical level from the conceptual level. The conceptual level or "software architecture" [5] comprises the artifacts themselves, their externally observable properties and the relationships among them.

Traditional SCM encompasses four main tasks [4][2]:

- **Configuration Identification**. This task uniquely identifies each component item of a software system, in order to enable accurate tracking and control of software changes. The concept of *configuration baseline* is used as the set of all documents prepared at a specified milestone in the software development process.
- **Configuration Control**. Administrative mechanisms and documents for evaluating and approving or disapproving proposed changes to a software system.
- **Configuration Accounting**. Records and reports information on the status of proposed changes and their implementation. The logs and reports generated by configuration accounting task serves administrative purposes and are used by the project personnel in the development process.
- **Configuration Audit**. This task serves two purposes: *functional auditing* and *physical auditing*. The first ensures that test and data analysis exists for every item and that the project team maintains technical information for each item. It validates the completion of every item in a satisfactory way. The second purpose is to provide guidance for the product configuration identification and for the identification of any differences between the physical configuration of items and those in the baseline.

Several tools soon appeared to deal with the high volume and complexity of the information manipulated in the course of these tasks, whose description falls beyond this text. However, it is important to describe the two main facilities present in all the tools [12]: **library management** and **change control**. Library management provides for classifying, storing and accessing the components of a software system, such as program design documents, source and object code files, user manuals and other related documents. The change control function manages the modification of a software system. It authorizes, controls and tracks component modification through the life cycle to implementation. All the tools fall in one of four models [9]:

- Check-in/Check-out
- Composition model
- Long Transaction model
- Change set model

The **check-in/check-out model** encompasses two kinds of tools: version control and (semi)automated build of the software system. SCCS (Source Code Control System) [10], RCS (Revision Control System) [21] and Microsoft Visual SourceSafe are examples of the first and *imake* [7] is an example of the second.

The **composition model** is a natural outgrowth of the check-in/check-out model. It relies on the notion of repository and work area, while providing for concurrency control through locking mechanisms. The major improvement over check-in/check-out model is the notion of configurations, entities understood by the SCM tool, allowing developers to compose a software system from its components and selecting a desired version for each component. A configuration consists of a *system model* and *version selection rules*. The versions of the whole software system can be modeled by means of an underlying component version graph (labeled version graph) or by predicates on attributed version objects (first order logic) [25]. Cooperative work is supported by sharing a work area, although concurrent changes over an item are forbidden. This model also

[1] A classic definition of Software Configuration Management can be found in [2] [4] [19] [21] as the set of activities necessary for the control of an evolving software system, such as identification and record of the consecutive stable stages of a software product and the artifacts needed for (re)building the product.

embodies the notion of *conceptual* or *logical change* as a formally approved change affecting several items and managed as a single entity.

Long Transaction model. The emphasis in this model is the support to the evolution of the software system as a whole. The system evolves by successive, apparently atomic, changes or *transactions*. This concept of transaction is analogous to database management systems transactions, although some differences apply. The SCM transactions are long and persistent. All the modifications to artifacts occur under the control of one transaction and the change is only visible after the transaction has been committed. Several concurrency control schemes can be attached to transactions in order to provide for team coordination of concurrent change. This model can be found in the AEGIS tool [15][16].

The **change set model**. The concept of *change set* [17] [24] is introduced in this model and represents the modifications in several components to achieve a logical change. In previous models the users only could operate with versions of components or versions of configurations. Using SCM tools supporting this model, developers can operate directly on change sets, track logical changes and determine whether these changes are part of a given configuration. In this model a configuration can be thought of consisting of a baseline and a combination of change sets.

Some attempts have been made to subsume all these models in one unifying system [25] [8]. Existing SCM systems and tools implements one or several of the above models. Work on SCM systems is underway for more than 25 years [8] but is still faced with some challenges that have to do with the supporting data model of the SCM, the way software systems are developed nowadays (component based development, web based cooperation, use of disparate programming languages, ...) and process management.

III. Case Study

An example may help to illustrate the above concepts and points of view. The case study briefly presented is a commercial Information System for shoe manufacturing industry called SOVELA[2]. The functional modules of this system are graphically depicted on Figure 3- . This system was developed back in 1990 and is in use in several shoe factories. Originally, it was developed in Informix 4GL; a SQL embedded programming language for Informix DBMS. Now is almost completely rewritten using an object oriented programming language: Visual Objects [23].

The system has been deployed in different configurations, sometimes with modules replaced by ready made modules from other software systems. Those SOVELA modules are then replaced by a data interface, or merely a data import/export mechanism, in order to provide for systems integration.

As with any other software system of medium size, SOVELA was developed and is maintained by a small team - 2 to 4 software engineers (the composition of the team has changed, with only one of the former developers left). Typical problems related with sharing artifacts by team developers and related with change management had to be addressed. Also, as the system has been installed in shoe factories with different work organization, it needs to support increasing amounts of variability.

The first SCM tools used by the developing team were SCCS and the Unix *make* utility. Now, Microsoft Visual SourceSafe is used and is available directly from the Visual Objects integrated development environment (IDE). Visual Objects IDE is repository based and supports the concept of software project. Unfortunately this repository is not team aware, so each developer has its own repository, importing shared artifacts to the individual workspace. There is a centralized SourceSafe database that controls the versions and the shared access to common artifacts thru a check-in/check-out mechanism.

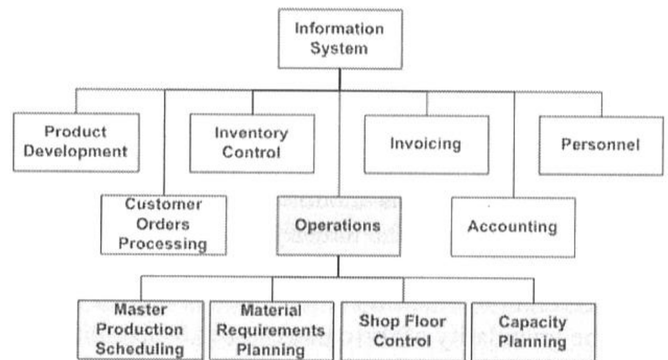

Figure 3- Enterprise Process Diagram

The SOVELA software system focuses on the support to operations management and job shop floor control.

From the beginning some variability was expected and anticipated. To some extent, domain analysis was exercised in order to accommodate in the initial design different ways of work organization. Later, several customizations for particular contexts were undergone. The Operations main module is the one that exhibits a higher amount of variability.

Roughly, the life cycle of a pair of shoes can be described in four phases: cutting, sewing, assembly and finishing. As an example, the sewing phase means to join together and sew several leather pieces of the uppers. Typically, this was (and still is) done by several workers - 10 to 40 on a production line, each performing a different sewing operation, on the same batch of shoes. Each worker has a small amount of pieces nearby to work with, usually a plastic box with a lot size of 10 pairs. The 10 pair's box moves from one worker to the next, as each sewing operation is finished on the lot. The boxes are carried by a conveyor belt.

[2] Sovela is the portuguese name for a tool, a piercer, used to sew by hand the leather uppers of shoes. Nowadays, it has been replaced by machinery.

Initially, two variants for work organization were foreseen, adequate for two different lot distribution approaches. The first one is based on the realization of the sewing operations in a strictly predefined serialized order. In the second approach a supervisor distributes the boxes to the workers, as they finish the lot in hand, and each worker is capable of executing more than one different operation. Both are adequate for mass production, typical in the shoe manufacturing industries.

These two approaches call for variants in the Shop Floor Control and Capacity Planning modules.

More recently, as the size of customer orders becomes smaller, a completely different work organization approach was undertaken by some shoe factories. Instead of a serial production line (straight-line production), the workers in the sewing section are divided into groups of 4 - 5 workers. The group is responsible for executing all the sewing operations for each pair of shoes, in successive 10 pair lots. SOVELA was then modified to support this new work organization. The changes to the software product were made on top of a working version. All the function modules under Operations had to be modified.

All three work organizations methods have different balancing problems, different coordination of the flow of materials and different scheduling criteria, just to name a few differences.

At the implementation level, these differences mean different input data, dissimilar computations and business rules, and different outputs. The overall architecture of the Operations module needs to have variants also. The schema of the underlying supporting database has also variants, with changing tables and attributes and different relationship among them, for each variant.

The granularity of the changes goes from entire modules to single lines of source code. Extensive refactoring of source code was (is) needed to achieve the changes.

IV. Proposal

A clear distinction must be made between the concept of versions of a software product and the mechanisms for tracking differences in text files. A well-known goal of SCM is to integrate all the dimensions of the evolution of a software product into an orthogonal version control system [25].

Three dimensions are considered (Figure 4-), for the evolution of a software system, with three axis representing the projection of the successive versions over time, space (logical) and cooperative work.

Historic versions – represents intermediate stages, stable and immutable, of the product, as it evolves over time.

Logical versions – when several different instances of the system exists at the same time, considering heterogeneous platforms or customizations for specific users.

Cooperative versions – represents cooperative development, with concurrent activities on shared artifacts.

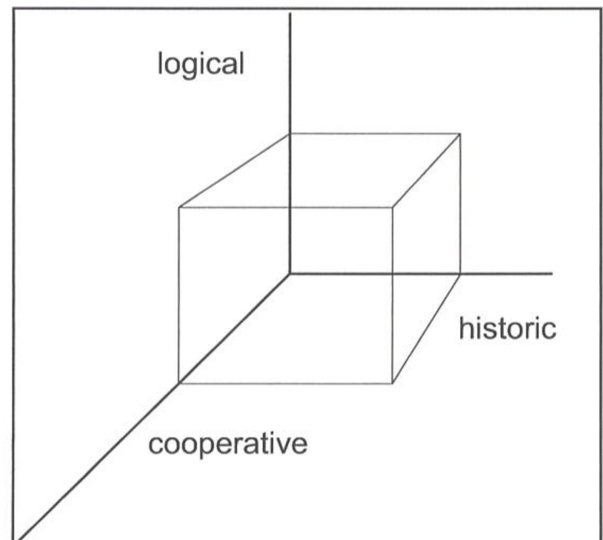

Figure 4- Three dimensions for evolution

Traditional SCM techniques and systems only handle the problems posed from the historic and cooperative evolution. The integration with the logical axis is not considered at all or is considered in a way that is incomplete. Sometimes the problem is avoided by splitting the system into two different ones, even if some common artifacts are inherited. Sometimes the evolution over the logical axis is treated as temporary stages, merged together in only one product as soon as possible. The solution for these problems may not be found in the classic approaches to SCM.

We think that the main issue is an adequate data model, capable of representing the software system as it evolves over the three dimensions. A formal model representation or new *architecture* is needed as the foundation for the SCM tools. The concept of software architecture has emerged quite recently [5] [8] [14] [6] [22] in the SCM research. The architecture, or "structure of structures", comprises software components, the externally perceived properties of those components and the relationships among them. Common software structures are already identified [5]; some of them are as follows.

Module structure. Units useful for work assignment and resource allocation, during the development and maintenance life cycle. Each module has well defined software artifacts (like source code files, interface specifications, test plans). The modules in this structure are related by relationships of the type "is a sub module of".

Logical structure. Represented by abstractions of the functional requirements [1], linked by "shares data with" relation.

Process structure. A perspective orthogonal to the preceding ones, deals with the aspects of running the software systems. The units are computer system resources dynamics (processes or threads). The types of relationships among them are "synchronizes with", "can't run without", "can't run with", "preempts" or other related with process synchronization and concurrency.

Physical structure. The physical structure reveals the mapping between software and hardware. It is more important in distributed or parallel systems. The components are hardware artifacts and the links are communication channels of type "communicates with".

Calls structure. The units are procedures or modules, related by "calls" or "invokes" relationships.

Control and data flow. The units are programs or modules with relationships among them such as "becomes active after" or "send data to".

Class structure. The units are objects and the relationships are "inherits from", "is an instance of" or "is composed by".

The notion of architecture shows early, in an implicit way, in the check-in/check-out model, historically the first one to appear. It was then used merely as a dependency identification mechanism between modules for compilation, commonly implemented as an input text file (*makefile*) to rebuild tools [7]. This concept is expanded in other rebuild tools for systems like PROTEUS [19], ICE [26] and the Ragnarok system model [6].

Other architectural approaches to deal with evolving software systems are Architecture Description Languages (ADL) [13] [18] and Software Patterns [1] [3][11].

V. Conclusion

Software Configuration Management was first associated with administrative tasks in the process of development of software systems. It evolved from simple tools to integrated software products aimed at aiding teams of developers building complex systems in shared environments. Still, it lacks some important features, essential to support the modifications over three dimensions - logic, historic and cooperative – when viewed as orthogonal paths for system evolution.

Traditional models (check-in/check-out, composition, transaction and change set), used as foundations for existing tools, were presented. These models and the SCM tools based on them are affected from the tight relation with physical aspects of underlying file systems and data representations.

A new data model representation for the software system architecture is needed, in order to overcome existing limitations. Some structures, that must be present in the model, and the relationships among them, are enumerated. Further proper formalization is, of course, necessary.

References

[1] B. Appleton, S. P. Berczuk, R. Cabrera, R. Orenstein, "Streamed Lines: Branching Patterns for Parallel Software Development", *Proceedings of Pattern Languages of Programs'98,* 1998

[2] S. Ayer, F. Patrinostro, "Software Configuration Management - Identification, Accounting, Control and Management", McGraw-Hill Inc., 1992

[3] K. Beck, W. Cunningham, "Using Pattern Languages for Object-Oriented Programs", Technical Report No. CR-87-43, 1987.

[4] Edward H Bersoff, Vilas D. Henderson, Stanley G. Siegel, "Software Configuration Management", Prentice-Hall Inc., 1980

[5] Paul C. Clements, "Rationalize Your Design. In Constructing Superior Software", MacMillan Technical Publishing USA, 2000

[6] H. B. Christensen, "The Ragnarok Architectural Software Configuration Management Model", *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, 1999

[7] P. DuBois, "Software Portability with imake", O'Reilly & Associates, Inc, 1996

[8] J. Estublier, "Software configuration management: a roadmap", *In A. Finkelstein, editor, "The Future of Software Engineering"*, Special Volume ICSE 2000, 2000.

[9] P. Feiler, "Configuration Management Models in Commercial Environments", Technical Report CMU/SEI-91-TR-7, SEI – Carnegie Mellon University, 1991

[10] B. W. Kernighan, "The Unix System and Software Reusability" in *IEEE Transactions on Software Engineering* SE-10 (5, September 1984): pp. 513-

[11] M. Grand, "Patterns in Java – A Catalog of Reusable Design Patterns Illustrated with UML", Volume 1, John Wiley & Sons, Inc., 1998

[12]C. McClure, "The Three Rs of Software Automation", Prentice Hall, 1992

[13] N. Medvidovic, D. S. Rosenblum, "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures", *Proc. First Working IFIP Conf. on Software Architecture*, February 1999, pp. 161-182.

[14]N. Mehta, N. Medvidovic, S. Phadke, "Towards a Taxonomy of Software Connectors", ICSE 2000, 2000.

[15]P. Miller, "Aegis - A Project Change Supervisor: User Guide", 2004

[16] P. Miller, "Aegis - A Project Change Supervisor," *AUUG '93 Conference Papers,* 1993, p. 169-178.

[17] Bjørn P. Munch [et al.], "Uniform Versioning: The Change-Oriented Model", Proceedings of the 4th International Workshop on Software Configuration Management, 1993.

[18] J. E. Robbins, D. F. Redmiles, D. S. Rosenblum, "Integrating C2 with the Unified Modeling Language", *Proceedings of the 1997 California Software Symposium*, November 1997, pp. 11-18.

[19] I. Sommerville, G. Dean, "PCL: A configuration language for modeling evolving system architectures", Lancaster University, 1994

[20] Ian Sommerville, "Software Engineering", Addison-Wesley Publishers Ltd., 1992

[21] W. F. Tichy, "Configuration Management", John Wiley & Sons, 1994

[22] T. Weiler, "Modeling Architectural Variability for Software Product Lines" *in Proceedings of the Software Variability Management Workshop, 13-14 February 2003. J. v. Gurp and J. Bosch:* pp. 53 – 61

[23] I. Wessel, G. Bless, "CA-Visual Objects Effective Programming", CARL HANSER VERLAG, 2003

[24] B. Westfechtel [et al.], "A Layered Architecture for Uniform Version Management",

[25] A. Zeller, G. Snelting, "Unified Versioning through Feature Logic", *in ACM Transactions on Software Engineering and Methodology 6(3)*, July 1997

[26] A. Zeller, "Configuration Management with Feature Logics", Technical Report - Technische Universitat Braunschweig, 1994