RESEARCH PAPER | Information Technology | Volume : 3 | Issue : 6 | June 2013 | ISSN - 2249-555X

Assessment of Versioning Paradigms with respect to Software Configuration Management

KEYWORDS Versioning, configuration, management.

Kirti Mathur
International Institute of Professional Studies
D. A. University, Indore

Amber Jain
International Institute of Professional Studies
D. A. University, Indore

ABSTRACT *Software still remains a major challenge since 50 years for theoretical and practical work in Software Configuration Management (SCM), tracking and controlling changes. This paper provides an overview with classification of different versioning models and paradigms by defining fundamental concepts such as revisions, variants, configurations, and changes. It assesses various versioning schemas used in different companies as commercial and open source systems. As a result of this survey, this paper then proposes Github's Semantic Versioning as the most consistent and logical to control versioning.*

II. Introduction

Software configuration management is the discipline of tracking and controlling changes in large and complex software systems. SCM practices include revision control and the establishment of baselines. If something goes wrong, SCM can determine what was changed and who changed it. The importance of SCM has been widely recognized, as reflected in particular in the Capability Maturity Model (CMM) [1] developed by the Software Engineering Institute (SEI) [9]. SCM is seen as one of the key elements for CMM. Furthermore, SCM plays an important role in achieving ISO 9000 conformance. SCM can serve both as:

- Management support discipline : SCM is concerned with controlling changes to software products such as identification of product components and their versions, change control, status accounting, and audit and review

- Development support discipline : SCM provides functions that assist developers in performing coordinated changes to software products. SCM is in charge of accurately recording the composition of versioned software products evolving into many revisions and variants, maintaining consistency between interdependent components, building compiled code and executables from source, and constructing new configurations based on project descriptions.

In this paper, we will consider SCM as a development support discipline.

III. LITERATURE REVIEW

In software engineering, software configuration management (SCM) is the task of tracking and controlling changes in the software, part of the larger cross-discipline field of configuration management [8]. The major goals of SCM are:

- Version and Configuration control
- Configuration identification
- Build management
- Configuration status accounting
- Defect tracking
- Environment management
- Process management
- Configuration auditing
- Team interactions and teamwork

This paper primarily focuses on overview and classification of different versioning paradigms and proposes SemVer as the consistent and logical versioning scheme.

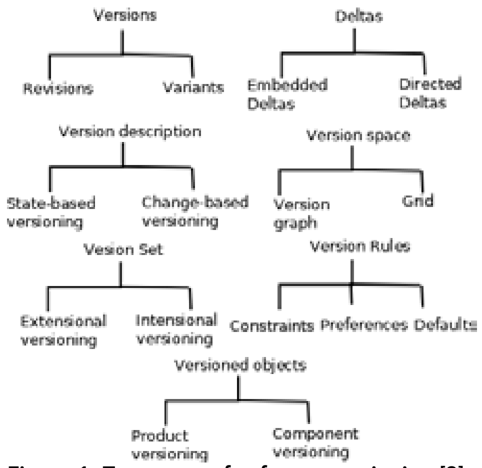

Figure-1: Taxonomy of software versioning [3]

A version model defines the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. Software objects and their relationships constitute the product space, their versions are organized in the version space. A versioned object base combines product and version space [16].

Many SCM systems use version graphs for representing version spaces. A version graph consists of nodes and edges corresponding to (groups of) versions and their relationships, respectively.

SCM system is illustrated in Figure, where versioned objects, revisions, and variants are organized into orthogonal dimensions:

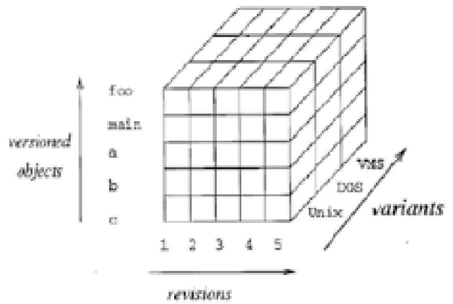

Figure-2: Software product versioning [4]

Volume : 3 | Issue : 6 | June 2013 | ISSN - 2249-555X

Several SCM systems are founded on databases and manage versions of objects and relationships stored in the database. Graphs are well suited to represent the organization of a versioned object base, even if the corresponding system is not graph-based. For example, SCCS and RCS are both file-based, but the version space of a text file may be represented naturally as a version graph [10].

To represent versions in the object base, deltas are being used both at the coarse- grained and the fine-grained levels. Deltas are mainstay in Version Control Systems and are calculated using some data differencing algorithms/programs (for e.g. diff utility [18]). Formally, a data differencing algorithm [17] takes as input source data and target data, and produces difference data such that given the source data and the difference data, one can reconstruct the target data.

These deltas can then be used by Merge tools to combine versions or changes . Most version control tools (such as Git, Mercurial, Subversion, CVS, bzr, SCCS etc [14]) implement deltas and merging to ease the management of versions or changes. Merge tools may be classified as follows (Figure-3: Types of merging):

1. **Raw merging:** This simply applies a change in a different context. For example in Figure-3, change c2 was originally performed independently of change c1 and is later combined with c1 to produce version v4. This is used by SCCS.

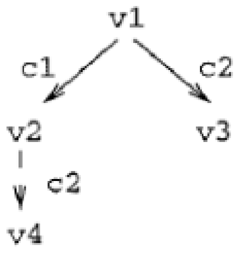

Figure-3(a): Raw merging [3]

2. **Two-way merging:** This compares two alternative versions a1 and a2 and merges them into a single version m. This can only detect differences, and cannot resolve them automatically.

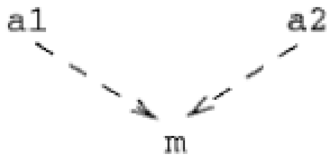

Figure-3(b): 2-way versioning [3]

3. **Three-way merging:** This consults a common baseline b if a difference is detected. If a change has been applied in only one version, this change is incorporated automatically. Otherwise, a conflict is detected that can be resolved either manually or automatically.

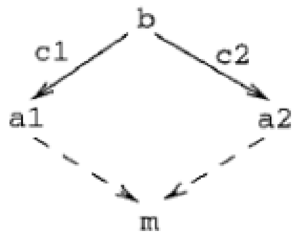

Figure-3(c): 3-way merging [3]

IV. Versioning Schemes

Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software [15]. A consistent and stable versioning scheme is one of the most important aspects of SCM. Many version numbering schemes are used to keep track of different versions of a piece of software:

Company	Versioning Scheme	Example
Microsoft	Change significance: · Emphasize the value of the upgrade to the software user. · Represent a release half-way between major versions	Internet Explorer 5.1.1
Microsoft	Designating development stage: · Releases that are not stable enough for general or practical deployment · Releases are intended for testing or internal use only · Alpha, Beta, Release Candidate etc.	Windows 7 alpha, Windows 8 beta, Windows Vista Release Candidate
Adobe	Number of sequences: · Fourth (usually unpublished) number which denotes the software build and/or build date.	Adobe Flash 10.1.53.64
Wikimedia	Incrementing sequences: · Free software packages treat numbers as a continuous stream.	MediaWiki 1.10.0, 1.11.0, 1.11.1, 1.11.2
Canonical	Date: · Uses the year and month (optionally followed by the day of the release).	Wine 20040505, Ubuntu 11.10
Adobe, Microsoft	Year of release: · Identify versions by year.	Adobe Illustrator 88, Microsoft Windows 2000 Server
Adobe, Macromedia	Alphanumeric codes: · Easier to read and refer to by customers.	Macrome-dia Flash MX, Adobe Photoshop CS2
Apple	NumVersion struct: · one- or two-digit major version · a one-digit minor version · a one-digit “bug” version · a stage indicator (drawn from the set development/prealpha, alpha, beta and final/release) · a one-byte pre-release version When writing, convention is to omit any parts after the minor version whose value are zero	Mac 1.0.2b12
Sun Microsystem, Microsoft	Internal version numbers: · Usually supplement external/public version numbers · More consistent version numbering rules	Java 1.5.0 (public version Java SE 5.0), NT 5.1 (publicly Windows XP)

Table-1: Different versioning schemes in use in different software companies [2][6][3][4]

IV. Need Of Logical And Consistent Versioning

Enterprise software is designed to take advantage of other software components that are already available (rather than reinventing the wheel), or have already been designed and implemented for use elsewhere. Dependency hell [5] (also known as DLL hell on Windows, Extension Conflict on Mac OS, JAR hell in Java and RPM hell on Red Hat Linux based systems), in which installed packages have dependencies on specific versions of other software packages, is a major frustration of software users. The dependency issues arise around shared packages/libraries on which several other packages have dependencies but where they depend on different and incompatible versions of the shared packages. If the shared package/library can only be installed in a single version, the user/administrator may need to address the problem by obtaining newer/older versions of the dependent packages. This, in turn, may break other dependencies and push the problem to another set of packages.

As a responsible developer you will, of course, want to verify that any package upgrades function as advertised. The most obvious (and easiest, though often ignored) solution to this problem is to have a strict, standardized, logical and consistent version numbering system. This is not a novel or revolutionary idea. In fact, most developers and software companies do something close to this already (as outlined in previous section). The problem is that "close" isn't good enough. Without compliance to some sort of formal specification, version numbers are essentially useless for dependency management. What you can do is let Semantic Versioning provide you with a sane way to release and upgrade packages without having to roll new versions of dependent package saving manpower, time and effort.

V. Semantic Versioning

Many researchers have proposed consistent versioning schemes [11][13]. Tom Preston Werner proposed a logical, strict and consistent set of rules and requirements as Semantic Versioning (SemVer) [12] for assigning version numbers. According to this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next (so that developers can handle the problem of dependency hell).

In Semantic Versioning, the developer first declares a clear and precise public API (which either may consist of documentation or be enforced by the code itself). Once the public API is identified, the developer communicates changes to it with specific increments to the version number. As an example, consider a version format of X.Y.Z (X = Major, Y = Minor and Z = Patch). Bug fixes not affecting the API increment the patch version (Z), backwards compatible API additions/changes increment the minor version (Y), and backwards incompatible API changes increment the major version (X). A detailed secification of Semantic Versioning is available at http://semver.org/. To be able to use SemVer, developers or company needs to declare that they are doing so (by linking to SemVer website/specification from project's documentation so that others know the rules and can benefit from them) and then strictly and consistently follow the SemVer rules.

VI. Conclusion

Even though developers and software companies use different versioning schemes, industry needs to agree on a consistent, logical and strict versioning scheme as a solution to the dependency hell problem. One strong candidate is SemVer which is clear, consistent and concise and is being used (with minor variations) by many companies and developers.

REFERENCE [1] Capability Maturity Model. (2013a, April 19). In Wikipedia, the free encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=Capability_Maturity_Model&oldid=549477725 | [2] Coding Horror: What's In a Version Number, Anyway? (n.d.). Retrieved May 1, 2013, from http://www.codinghorror.com/blog/2007/02/whats-in-a-version-number-anyway.html | [3] Conradi, R., & Westfechtel, B. (1997). Towards a uniform version model for software configuration management. In R. Conradi (Ed.), Software Configuration Management (pp. 1–17). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/3-540-63014-7_1 | [4] Conradi, R., & Westfechtel, B. (1998). Version models for software configuration management. ACM Comput. Surv., 30(2), 232–282. doi:10.1145/280277.280280 | [5] Dependency hell. (2013, April 22). In Wikipedia, the free encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=Dependency_hell&oldid=551704064 | [6] Fogel, K. (2005). Producing Open Source Software: How to Run a Successful Free Software Project (1st ed.). O'Reilly Media. | [7] History of software configuration management. (2013a, March 16). In Wikipedia, the free encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=History_of_software_configuration_management&oldid=544680084 | [8] Humphrey, W. S. (1989). Managing the software process. Addison-Wesley. | [9] Paulk, M. C. (1995). The capability maturity model: guidelines for improving the software process. Addison-Wesley Pub. Co. | [10] Pressman, R. (2009). Software Engineering: A Practitioner's Approach (7th ed.). McGraw-Hill Science/Engineering/Math. | [11] Schmidt, E. E., & Lampson, B. W. (1985, December 10). Software version management system. Retrieved from http://www.google.co.in/patents?id=pKgsAAAAEBAJ | [12] Semantic Versioning 2.0.0-rc.1. (n.d.). Retrieved May 1, 2013, from http://semver.org/ | [13] Sigal, A. D., Bien, D., & Pissarra, A. (1999, March 9). Dynamic versioning system for multiple users of multi-module software system. Retrieved from http://www.google.co.in/patents?id=3gYXAAAAEBAJ | [14] Sink, E. (2011). Version Control by Example (1st ed.). Pyrenean Gold Press. | [15] Software versioning. (2013, April 24). In Wikipedia, the free encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=552018364 | [16] Sommerville, I. (2010). Software Engineering (9th ed.). Addison-Wesley.