Evaluating Complexities in Software Configuration Management

Frank Tsui, Southern Polytechnic State University, 1100 S. Marietta Parkway, Marietta, GA, 30060, USA; E-mail: ftsui@spsu.edu
Orlando Karam, Southern Polytechnic State University, 1100 S. Marietta Parkway, Marietta, GA, 30060, USA; E-mail: okaram@spsu.edu

INTRODUCTION

In software engineering, software configuration management tools such as Apache Ant, CVS or ClearCase [1, 3, 5, 9] are often included as an integral part of constructing large information systems or managing changes in information systems [7, 8, 11]. It is often assumed that the organizations involved in the development and support of information systems have naturally embraced the concept of software configuration management. While enterprises engaged in medium to large size information systems development and support do subscribe to the concept of configuration management, many smaller establishments only pay lip services to this important activity. In this paper we will explore the reasons behind this through analysis of levels of complexity in software configuration management (SCM). First SCM will be discussed, categorized and divided into four different dimensional areas. Then a set of volume metrics related to these dimensional areas will be defined. Levels of complexities of SCM, in terms of volume metrics, will be explored. We will utilize a real case of a software application development to demonstrate the utility of these metrics and how the levels of complexity of SCM may be used to help the decision process of incorporating SCM and SCM tools. Ultimately, our goal is to provide a clear measure of the degrees of SCM and an ordering scheme of implementing SCM.

SOFTWARE CONFIGURATION MANAGEMENT

Configuration management initially started with the management of pieces and parts. In software systems this often meant the management of files. As software systems became more complex and larger in size, the number of files and the structure that needed to be placed on top of the files had to be managed. Also, software systems became more expensive, and the life span of a software system extended into multiple years after its initial release. The large number of changes to the software system and the lengthy period of maintenance cycles of a software system needed some form of change management. This precipitated the inclusion of change control as an essential component of software engineering. SCM, as a discipline of managing parts and managing changes, started to grow in scope. It is an integral part of the software processes described by Software Engineering Institute [10]; however, it continues to be a domain of software engineering that is understood by a relatively small number of information and software engineering experts.

A software configuration management system provides a wide range of functionality. Dart [6] first classified this range of concepts into fifteen areas as follows:

- Repository
- Distributed component
- Context management
- Contract
- Change request
- Life-cycle model
- Change set
- System modeling
- Subsystem
- Object pool
- Attribution
- Consistency maintenance
- Workspace
- Transparent view
- Transaction

Not all of these functionalities are implemented by any single SCM tool. These functional areas are inter-related in serving four critical dimensions of software configuration management [4,11].

- a mechanism that describes the artifacts that will be managed
- a mechanism to capture, access, and control the artifacts
- a mechanism to construct a specific product out of the artifacts
- a mechanism that describes the relationship among the artifacts

Artifact Identification

In order to manage a large number of pieces of software artifacts, we must be able to identify and specify those artifacts that are produced through the development and support activities. The decision of which artifacts need to be managed is based on the project and the process. If the deployed process of the software project states that only executable source code is of importance and that is the only artifact type that needs to be managed, then we only need to label code pieces and manage the changes to the code. On the other hand, if other artifacts from the requirements, design, and test phases are considered important, then the mechanism must include all of them. The mechanism must be able to identify and specify the artifacts within each artifact type. In addition, a specific piece of artifact, regardless of type, may experience several iterations of changes. In order to control changes, each version of the changes may need to be kept. Thus, the artifact identification mechanism must be able to allow different level of sophistication, which is in turn dependent on the over all software process employed.

Let A be the set of artifacts that the software project process has determined to manage. Then for a specific artifact x in A, there needs to be at least three attributes: name, version and type. Thus for x, the three attribute components formulate a unique identifier as follows.

artifact identifier = name **.** version **.** type

Name may be a string of characters of some predetermined length. Version may be an integer of some predetermined number of positions. Type may be a two position code to identify artifact types such as requirement, design, logic code, screen code, data base table, help text, test case, etc. The symbol, ".", separates the three components of the identifier.

Artifacts Capture and Control

After each piece of software artifact can be uniquely identified, it still needs to be managed. There are two components to this dimension. First, all the artifacts must be captured. This is a fundamental activity of configuration management. If there is no one place where all the pieces and parts are kept, then assembling and building a system would be left to a high degree of chance of failure. Something inevitably is lost at the worst time such as the night before the software product release. The larger is the number of individual pieces of artifacts, the greater is the opportunity to lose something.

The second part is the access and control of the artifacts. There is rarely a situation where nothing is changed. Practically every type of artifact in software development and support experiences some degree of change. These changes must be conducted under a controlled process or the software system will quickly degenerate into a non-manageable system. The degree of control required depends on several parameters:

- number of artifacts under configuration management
- the anticipated amount of changes
- the number of people involved in the project
- the geographical and time distribution of work efforts related to the changes

Check-in and Check-out [1,3,5,9] are the two most often mentioned functions related to the access and control of artifacts. Check-out is the retrieval function. Except for security reasons, all artifacts may be retrieved. If an artifact is retrieved for the purpose of viewing, then another function, such as View, may be used. However, if an artifact is retrieved with the intent of performing a change to it, then it must be retrieved with the Check-out function. This is so that any conflict from multiple changes to the same artifact can be controlled. An artifact which is Checked-out is balanced with a Check-in of that artifact. An artifact that is currently Checked-out may not be Checked-out by another party until it is formally returned through Checked-in. Once a Checked-out artifact is updated through a Check-in, then essentially a new version of that artifact is formed. Thus the Check-out and Check-in pair of mechanism, along with version update, not only controls multiple changes but also keeps a history of the changes. Beyond this pair of basic control function, there are many other functions, such as compare or version incrementing, that exist to support the control mechanism. The amount of capture, access and control functionality needed, again, depends on the project.

Construct or Build

It would be somewhat pointless to have all the pieces identified, collected and put under control unless we are able to build a final software system that executes correctly. The construction activity is sometimes known as the Build. The simplest Build is the compile and link of a single module. Most of software systems are composed of a number of artifacts that require a much more complicated, multiple statements direction which includes the following information.

- the files which contain the different sources for compilation
- the target files for the results of compilation
- the different files required for the activity of linking
- the target files for the results of linking

More formally, the Build process may be described as two levels of relations, R1 and R2.

R1 is the relation that describes where the identified artifacts are stored and can be accessed.

R1 = A x F where, A is the set of identified artifacts and F is the set of folders or libraries where these artifacts are stored

R2 is the relation that maps R1 into steps of compile and link activities. The specific numerical order is important here. Thus it is defined as follows.

R2 = R1 x N where, R1 is defined as above and N is the natural numbers, which serve as an ordering mechanism

Thus the relation, R2, may be viewed as a sequence of steps in the build.

Software code Build is composed of and dependent on how well the two relations, R1 and R2, are constructed.

The time for a code Build cycle is directly related to R2, which is the sequence of steps to copy, compile, and link the code. Often times the Build cycle for a large, software system requires several mid-way interruptions and attempts to correct errors due to complexity of R1 or R2.

A comprehensive Build for a complete software product that includes the construction of executable code and of non-executables, such as User Guide or Read Me First notes, today requires multiple tools and different methodology. There does not exist one Build tool that can construct multiple artifact types. In order to perform such a complex Build, the SCM system must include the capability to handle, not just multiple versions of artifacts, but also relationships among multiple artifact types.

Artifact Relationships

With very small software projects, there may not be a complicated relationship among the artifacts. However, even with managing just one type of artifact, such as code, we need to account for the pieces of source code and the pieces of executable code. The source code is the artifact developed by the "coders". The executable, on the hand, is the post-compiled code. In order for the developed software to execute, often it requires the use of many other existing components. The obvious ones are the underlying operating system, the data base system, and the network system. In addition, there may be executable code from system and private libraries that must be included for the developed source code to compile and execute properly. Thus even within the code artifact type there may need to be a further differentiation of sub-types of code.

For very large projects where the process dictates that multiple types of artifacts are needed, two types of relationship within the project need to be considered.

- Intra-artifact relationship and
- Inter-artifact relationship

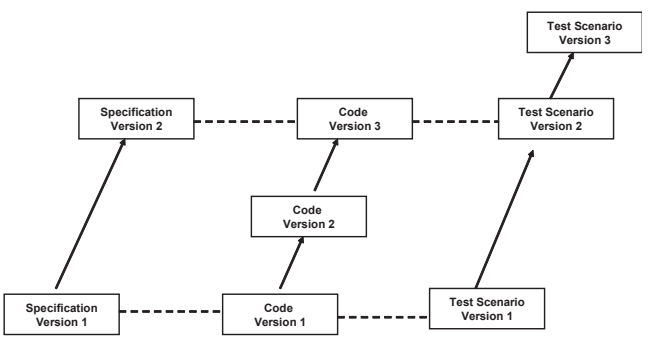
The intra-artifact relationship defines the relationship of the pieces within an artifact type. In the case of the executable software code artifact, the intra-relationship is stated in a set of statements related to the compilation and linking of the source code and reuse of other code in different libraries. This is a relatively simple software build process. If we require the use of other executable code such as a Tomcat [2] middleware or a specific database, then those executable code libraries must also be included in a larger build process where there is still a single artifact type but a large number of artifacts residing in different places.

The inter-artifact relationship defines the connections among different artifact types such as a requirement specification text, a source code which implemented that requirement, and a test scenario to test that implemented source code against the requirement specification. The relationship among these three types of artifacts may be further complicated when we introduce versions of the changes within each artifact type. See Figure 1.

Note that in Figure 1, the inter-artifact relationship among the specification, code, and test scenario artifacts are represented with dashed lines. The intra-artifact relationship is shown with solid arrows. There are two versions of specifications, three versions of code, and three versions of test scenarios. Associated with the first version of specification are version 1 and version 2 of code and version 1 of test scenario. The reason behind having two versions of code may be due to some error correction made to version 1 of code after conducting a test with version 1 of test scenario. Thus version 2 of code is the most updated version related to version 1 of specification and version 1 of test scenario. When the specification is updated to version 2, a code change is made and the related code is vesion3. The test scenario is updated to version 2 to reflect the corresponding changes made to version 1 test scenario. It is possible that the test scenario version 2 had an error and required a further update to create a version 3 test scenario. Thus specification version 2, code version 3 and test scenario version 3 form another inter-relationship among these three artifact types.

Keeping and maintaining a web of these relationships for a large software project can quickly turn into a nightmare. As the degree of complexity of inter and intra

Figure 1. Inter-relationship and intra-relationship



artifact relationships increases, an automated tool to help manage these relationships would definitely be a plus.

An ideal software product build would need to extend the current code Build. As such, the set of artifacts, A, in R1 would need to include all types of artifacts. R1 may be expanded to R1'. First define A_{tn} and $R1'_{tn}$.

A_{tn} = set of artifacts of type n

$R1'_{tn} = A_{tn} \times F$

Then, the extended R1, which includes more than one type of artifacts, is defined as:

$R1' = <R1'_{t1}, R1'_{t2}, - - -, R1'_{tn}>$

The activities of compile, link, merge, etc. depending on the artifact type, A_{tn}, for the second component of universal build would be defined as follows.

$$R2' = \{ R1'_{t1} \times N, \; R1'_{t2} \times N, \; \ldots \; R1'_{tn} \times N \}$$

Thus a general software product Build, which includes multiple artifact type relationships, is composed of R1' and R2'.

METRIC FOR SOFTWARE CONFIGURATION MANAGEMENT

In this section some basic metrics that applies to the four major dimensions of SCM will be introduced. The first metric gauges the volume of software artifacts that needs to be managed. Thus it impacts the dimensions of i) artifact identification and ii) artifact capture and control. SCM volume is an accumulation of all the uniquely identifiable artifacts. The SCM artifact volume, AV is defined in terms of the components of the artifact identifiers: name, version and type.

$$AV = \sum \text{unique artifact} = \sum \text{type} \sum \text{version} \sum \text{name}$$

Note that for each artifact name within an artifact type, there may be different number of versions. Pick the artifact name, across all the artifacts, which has the largest number of versions. Let that version number be version-max. Then the volume of the software product is bound by AVmax.

$$AV \leq AVmax = (\#\text{ of types}) * (\text{version-max}) * (\#\text{ of names})$$

The second metric is associated with SCM build. The normal code Build deals with just the single artifact type, code. Code build volume, CBV, may be measured in terms of R1 and R2. The ideal software product build may handle multiple artifact types. Thus SCM build volume for the ideal build, IBV, may be measured in terms of R1' and R2'.

First we define CBV as composed of two volumes, a) VR1, volume of R1 and b) VR2, volume of R2. Assume an element, a, to be an artifact of code type, and f to be an element of folders or libraries.

CBV = (VR1, VR2) where
VR1 = # of pairs in A x F = $\sum$ (a, f)
VR2 = # of steps in the sequence {R1 x N}

Note that these two volumes, VR1 and VR2 can not just be arithmetically added together to give CBV a single number because they are two different units. VR1 is pairs in A x F, and VR2 is elements in a sequence. IBV is also defined in terms of its components, R1' and R2'.

IBV = (VR1', VR2') where
$VR1' = \sum R1'_{tn} = \sum (A_{tn} \times F) = \sum \text{type} \sum (a_{type}, f)$
$VR2' = \sum \text{type} \sum (R1'_{type} \times N)$

Thus IBV is a pair composed of a volume, VR1', and VR'2. VR1' is the sum of pairs of (artifact, folder) across all the artifact types that is to be built, and VR2' is composed of number of steps in a sequence of build activities for each artifact type summed across all artifact types included in the software product build. Thus both CBV for code and IBV for more general build may be used as metrics for the dimensionalities of iii) artifact build of SCN and iv) artifact relationships of SCM.

UTILITY OF SCM METRICS

In this section we will describe our experience with a small application software project and the utility of these metrics in the decision process of whether an SCM tool needs to be brought in.

The application software project was initiated in 2002 to automate the graduate admissions process for three graduate departments. There are three major functional areas in the application software. Initially, requirements were collected and documented. Several rounds of requirements modifications were incorporated and a final specification document was produced. The developers then took over and the product was constructed and tested. All the discovered defects were fixed and the product was released with a 2 weeks period of product support by the original developers. The project statistics are as follows.

- Duration: 4.5 months
- # of people: 14 (part-time)
- People effort: 1344 person hours
- Major Artifact Types: Requirements Specification, Code, Test Cases, Test Reports
- Build Artifacts: 20 Java code files, 27 JSP code files for screens, 19 relational tables
- System platform: Apache Tomcat, Microsoft Window, Access DB

The initial developers managed the requirement specification as one evolving artifact, using an excel spreadsheet to track the major functional requirements. The requirements were collected by multiple persons, but the actual authoring of the specification was performed by one person. The implementation team divided the work among 1) data base code, 2) screen code and logic code by functional areas and 3) a control logic flow code. Essentially, the design and coding efforts were carried out pretty much together by the implementers, and the unit tested versions of the code were all submitted to one person who replaced existing code with new submissions. Thus only one version of code was ever kept.

The Build activity included only one artifact type, code. Only one final version of the requirements was kept and that related to the entire set of code artifacts. Test cases were kept but not controlled. Thus, code version is 1, and there were a total of (20 + 27 + 19) or 66 unique code artifacts.

$$AV = \sum \text{type} \sum \text{version} \sum \text{names} = \sum \text{names} = 66$$

CBV = (VR1, VR2) = (18 , 44) where
VR1 = 18 pairs of (a, f)
VR2 = 44 steps in the sequence of instructions

The application software product was essentially composed of the requirement document and one final version of code artifacts. So there was no reason to consider artifact relationship. The software product build is the same as code build, and IBV = CBV. For this level of complexity, the SCM utilized was an accounting of a list of artifact names and a code build with Apache Ant tool [1].

The application software has gone through two more rounds of modifications. Although several code artifacts were modified, no new code artifact was added to

the software application system. Since the newest version just replaced the existing version, no change history was kept. After two years and two maintenance cycles, the SCM metrics remained the same, with AV = 66, and CBV = (18 , 44).

Now the software application is in its fourth year and there is a large set of new requirements. The new software project will involve more than just minor modifications. First the current running system must be kept running, and a duplicate but completely separate application system needs to be made. Thus there will be 2 versions of all the code artifacts. The new set of requirements will be associated with the second code version, and the old requirements of 4 years ago will be associated with the existing code version. The large set of new requirements is expected to add some new code artifacts and modify some old code artifacts. Thus the new AV is expected to be much larger than 66. The CBV for the new project may not increase much beyond the current CBV. However, because of the need to maintaining two versions, each associated with a different set of requirements document, we have to consider 2 IBVs. For IBV of the original application software system, the VR1' is the same as adding 1 requirements artifact in a separate library or one new pair (a, f) to the old VR1. Thus VR1' = 18 + 1 =19. Similarly, VR2' just include one more instruction step to build the requirements document. So VR2' = 44 +1 = 45. Since we do not expect too much change in the build instruction, the IBV for the second version may not differ much from the IBV of the first version. The big difference this time can be summarized as follows.

1. A large increase in AV is expected for the second version of application of software.
2. Since there will be two application versions, the original AV is still there.
3. There will be an association of requirements to code. Thus there will be two IBVs.

Even though the actual software project is still fairly small as the original version and no increase in development complexity is anticipated, we are now maintaining two versions of software products. This increased complexity of SCM is forcing us to consider the incorporation of additional SCM tools.

CONCLUDING REMARKS

We have introduced a set of volume metrics, AV, CBV, and IBV to gauge the SCM complexity. We have also found that when AV is small and only CBV is involved, the SCM complexity can be managed with minimal tool. But once we grow the AV and start to deal with multiple IBV metrics, it is an indication to start considering sophisticated SCM tools.

An area for future extension is to investigate the volume metric for managing the impact of changes and the impact of building non-code artifacts. This would take the IBV to another level of measurement.

REFERENCES

1. Apache Ant, http://ant.apache.org, 2006
2.Apache Tomcat, http://tomcat.apache.org, 2006
3. Atria Software, ClearCase User's Manual, Natick, MA, 1992.
4. M.E. Bayes, Software Release Methodology, Prentice Fall, 1999.
5. CVS, www.gnu.org/software/cvs, 2006.
6. S. Dart, "The Past, Present, and Future of Configuration Management, SEI Technical Technical Report, CMU/SEI-92-TR-8, 1992.
7. J. Estublier, et al, "Impact of Software Engineering Research on the Practice of Configuration Management," ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 4 October 2005, pp 383-430.
8. A. van der Hoek, et al, "A Testbed for Configuration Management Policy Programming, IEEE Transactions on Software engineering, vol. 28, No 1, January, 2002, pp 79-99.
9. IBM Rational ClearCase, www-306.ibm.com/software/rational/offerings/scm.html, 2006.
10. SEI, www.sei.cmu.edu/legacy/scm, 2006.
11. F. Tsui and O. Karam, Essentials of Software Engineering, Jones and Bartlett Publishers, 2006.