Configuration Management for Reusable Software

William B. Frakes
Computer Science Department
Virginia Tech
wfrakes@vt.edu

Abstract

This paper discusses the configuration management of reusable software, and proposes an architecture that incorporates configuration management with a software library.

Introduction

Software configuration management concerns monitoring and controlling changes to software. This paper discusses configuration management of reusable software assets, drawing on previous work on configuration management for traditional software engineering [Frakes et. al. 91]. Configuration management has three major activities:

• *Version control.* Reusable software components, like any software product, will have versions because of error fixes and enhancements. To build a system using these assets, one needs to know which version to use. Old versions of assets must be recoverable for reference, and so they can be used to make corrections and enhancements. As software assets change, they form successive versions. Version control is the activity of keeping track of these versions.

• *Change control.* Change control is the procedure for requesting changes, deciding what changes to make, making changes, and recording and verifying changes. Changes to reusable assets in a library cannot be made haphazardly, but must be made under a controlled process.

• *Build control*. Keeping track of which versions of work products go together to form a release, and generating derived assets and systems correctly, is called build control. Build control for reuse has two aspects. One is the general specification of which versions of assets to use in a system build. The other aspect is that reusable assets may themselves be composites of other items, so specifications of how to build assets may also be required.

We discuss each of these activities in turn as it relates to reusable components..

Version Control

A *software configuration item* is any software project work product treated as a unit for version control. Examples of reusable assets that might be put under version control include functions and subroutines, classes, requirements and design documents, sets of test cases, header files, and user documents. Assets are often changed producing variants. Variants are sometimes distinguished as *versions* to be kept track of in the version control system. The first version of an item is called a *baseline.* For example, a software component might be baselined at the end of testing. A second version might be generated if errors are discovered after field release, a third version when the component is enhanced with new functionality.

Except for baselines, versions are always created by changing previous versions, called *predecessors*. A version is a *successor* to its predecessor. There are two kinds of versions:. A version is a *revision* if it replaces another version. A version is a *variation*, or *branch*, if it is one of several alternative versions. Variations may be created from other versions, or may be a baseline. Revisions are produced because it is constantly necessary to correct errors, remove or add information, and so forth. Variations are generated because of the need to tailor components to different environments and users.

To illustrate, suppose that a baseline component is changed to correct several mistakes. The new version is a revision of the baseline because it is meant to replace it. Two additional components might be generated from the corrected version describing two slightly different components for different operating systems. Because these three components are alternatives, with none meant to replace the others, they are variations, not revisions. In large projects generating multiple releases or multiple variations of releases for different operating environments or markets, there may be many variations and revisions of all items. Version control systems must keep track of all versions of items, including successor and predecessor information.

The design of reusable assets involves the prediction of variant future uses, and reusable assets should be designed to make modifications as easy as possible. In theory, the changeable parts of an asset should be hidden from a user; the visible interface should not change or should change minimally when errors are corrected or enhancements made. Thus, the interface and implementation of an asset might be treated as distinct software configuration items.

Version control is primarily record keeping, and a database tool is often used to support it. In a small ad-hoc reuse environment an informal system might do. Usually, however, the task is large and complex, so mechanized tools like Source Code Control System (`SCCS`) [Rochkind, 75]and Revision Control System (`RCS`) [Tichy, 85] are required.

Change Control

Change control is the activity of considering, deciding on, delegating responsibility for, and monitoring changes to items. Change control is instituted as a collection of procedures, usually focused on a database system for documenting and tracking changes.

Such a change tracking database system is called a *change control system*. Change control may be informal on small projects, perhaps using a logbook to document and track changes. On larger projects, strict change control procedures must be instituted along with a mechanized system for documenting and tracking changes.

The basic activities and processes of change control are the following: 1. Someone submits a request for a change. Such requests are called MRs (modification requests), and submitting a request is called *opening* an MR. Usually a form, either paper or electronic, called a *modification request form* or MRF is used to open an MR. An example of an MRF follows:

```
Modification Request:   Date of MR:
APPLICATION: TYPE: SW RELEASE OCC:
ORIG. NAME: EMAIL:
DESCRIPTION:
RELATED MRs:
STATUS:
```

```
PRIOR STATUS:
DUE DATE:
TESTER:
CATGORY CHNG:
RESOLUTION SUMMARY:
RESOLUTION:
IMPACT:
CHILDREN:
```

MRs may be opened by customers, developers, managers, testers, documenters, or some selected subset of these groups. MRs may mention faults, errors, enhancements, typos, and so forth.

If an MR is written against a requirement during the latter part of the life cycle, the change will also possibly require changes in the design, code, test plans, and other later life-cycle products. The MRs written to change these later life-cycle products are said to be *child MRs* of the original MR. Another example of child MRs are changes in individual compile modules dictated by a change in a common header file. Here, MRs to change the compile module are child MRs of the MR for the header. Reusable components will also be subject to these concerns.

2. MRs are reviewed by a *change control authority* or an *MR review board* that examines each MR and decides on an appropriate action. The MR review board may be composed of developers, managers, testers, and so forth. In a typical development environment, the MR review board will be project specific and may last only for the length of the project. In a reuse situation, the authority of the MR review board may have to extend across projects and organizations, and will have to last the lifetime of the reuse collection.

Often the decision involves classifying the MRs according to their severity. A classification system like the following is sometimes used:

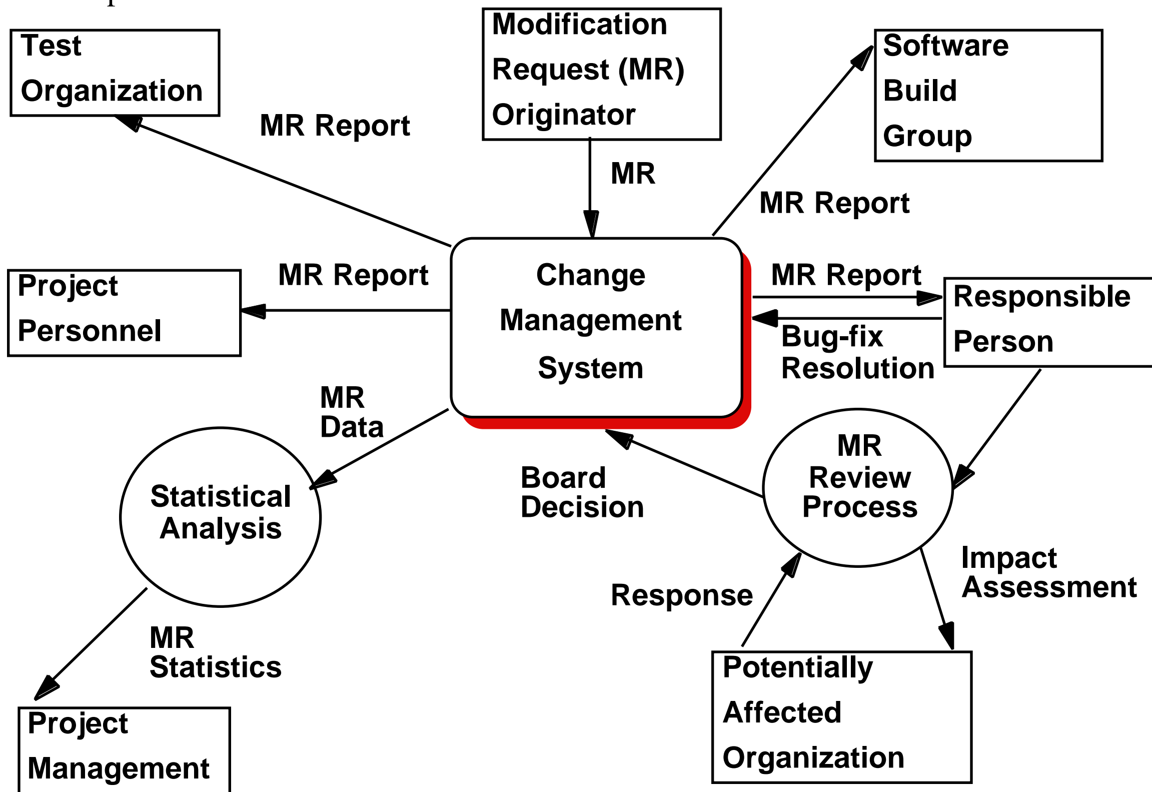
- *Severity level 1*. The item is unusable, incomprehensible, or unmanufacturable because of a problem reported by the MR. For example, an MR reporting a software fault causing the system to crash during typical use would rate a severity rank of 1.
- *Severity level 2*. The item is usable but unacceptable because of the problem reported in the MR. For example, a test case with partially incorrect input conditions might be given this severity classification.
- *Severity level 3*. The item is usable, but the MR reports a failure to conform to standards, guidelines, or practices. For example, an MR reporting improperly formatted section headings in a user document would receive a Level 3 severity ranking.
- *Severity level 4*. The MR requests an enhancement or adaptation. For example, a request for a port to a personal computer would rank as a severity Level 4 MR.

Once MRs are classified, the MR review board decides how to dispose of each MR. Among the possibilities accepting the MR and making the required change immediately, deferring the MR, or rejecting the MR .

3. Once a change is approved, a *change tracking authority* is notified and proceeds to track the progress of the change. `SCCS` provides crude support for build control of non-derived items in its version numbering system.

4. The change must be planned, scheduled, assigned to a developer, implemented, reviewed, and verified. Once this is done, the change control authority is notified that the change is completed. The MR is *closed*, and the change becomes part of the next version of the software configuration item. Change control systems can usually report the status of MRs,

generate statistics and reports summarizing the change activity of a project, and direct notifications and information about changes to people. Figure 2 illustrates the change control process.



Formal change control systems introduce significant management overhead in development and maintenance. Consequently software configuration items are usually placed under change control at the same time they are placed under version control, usually as late in the life cycle as possible.

Build Control

A *software configuration* is a set of item versions. Build control is the activity of specifying, tracking, and forming software configurations. Build control centers around the database activity of maintaining complete specifications of software configurations as items change thorough the life cycle.

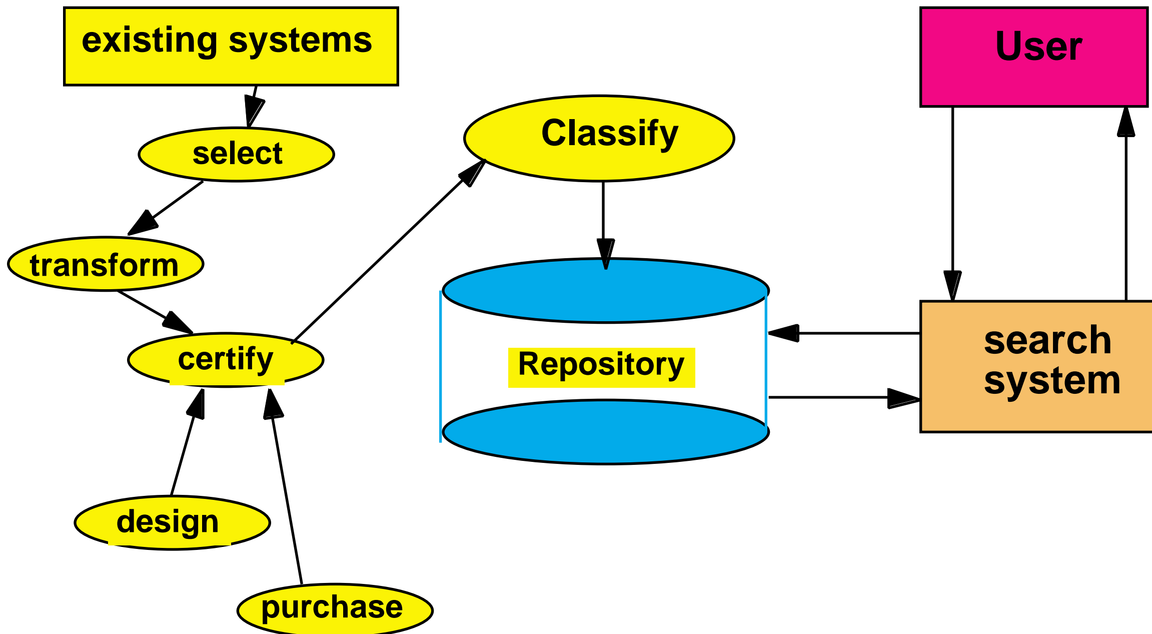
An important group of items that are part of software configurations are *derived items*, items generated from other items, usually by the computer. Examples of derived items include object modules, executable files, test data, and so forth. Non-derived item build control is not difficult until there is more than one release of a system; then it can become a difficult problem. Non-derived item build control is usually done by hand. Efficient derived item build control is a difficult problem, but it yields to mechanization; many tools for derived item build control exist. For example, the primary build control tool in the UNIX environment is **`make`**, which automates the generation of derived items.

Besides the primary activities of configuration management, configuration management tools can produce data to help with project tracking. For instance, **`SCCS and RCS`** can report how many lines have changed in project files. Since these changes, or *deltas*, are

easier to obtain than corrected error data, they are sometimes used as proxy metrics to estimate fault densities. Such data can be useful as an indicator of component quality.

Software Reuse Libraries

A reuse library consists of a repository for storing reusable assets, a search interface that allows users to search for assets in the repository, a representation method for the assets and facilities for change management and quality assessment. Surveys of reuse libraries may be found in [Frakes&Gandel 90] and [Milli et. al. 98].



Software Library

The figure above shows the major features of a reuse library. Reusable assets for the library can be obtained through the re-engineering of parts in existing systems through a process of selection and transformation or they may be designed from scratch or purchased. However obtained, the assets will need to go through a certification process to assure that they meet the quality standards required by the library. Studies have shown that if users believe that the library contains substandard components they will avoid using it [Carle 87].

As shown in table 1, a distinction is often drawn between different levels of published literature and this distinction applies to software assets as well as traditional printed media such as books and papers. The primary reuse literature consists of the assets themselves, for example, code modules. The secondary literature consists of indexes which guide a potential user to the primary literature. The tertiary literature provides an index to asset indexes and so on. The process of developing good secondary and tertiary indexes for reusable software is ongoing. One important development has been the identification of a link between indexing and domain analysis [Frakes et. al. 98].

Literature Level	Book	Asset	Example
Primary	book	asset	code module
Secondary	index to books	index to assets	index of code

			modules
Tertiary	index to index to books	index to index to assets	index of indexes of code modules

Table 1: Levels of Indexing

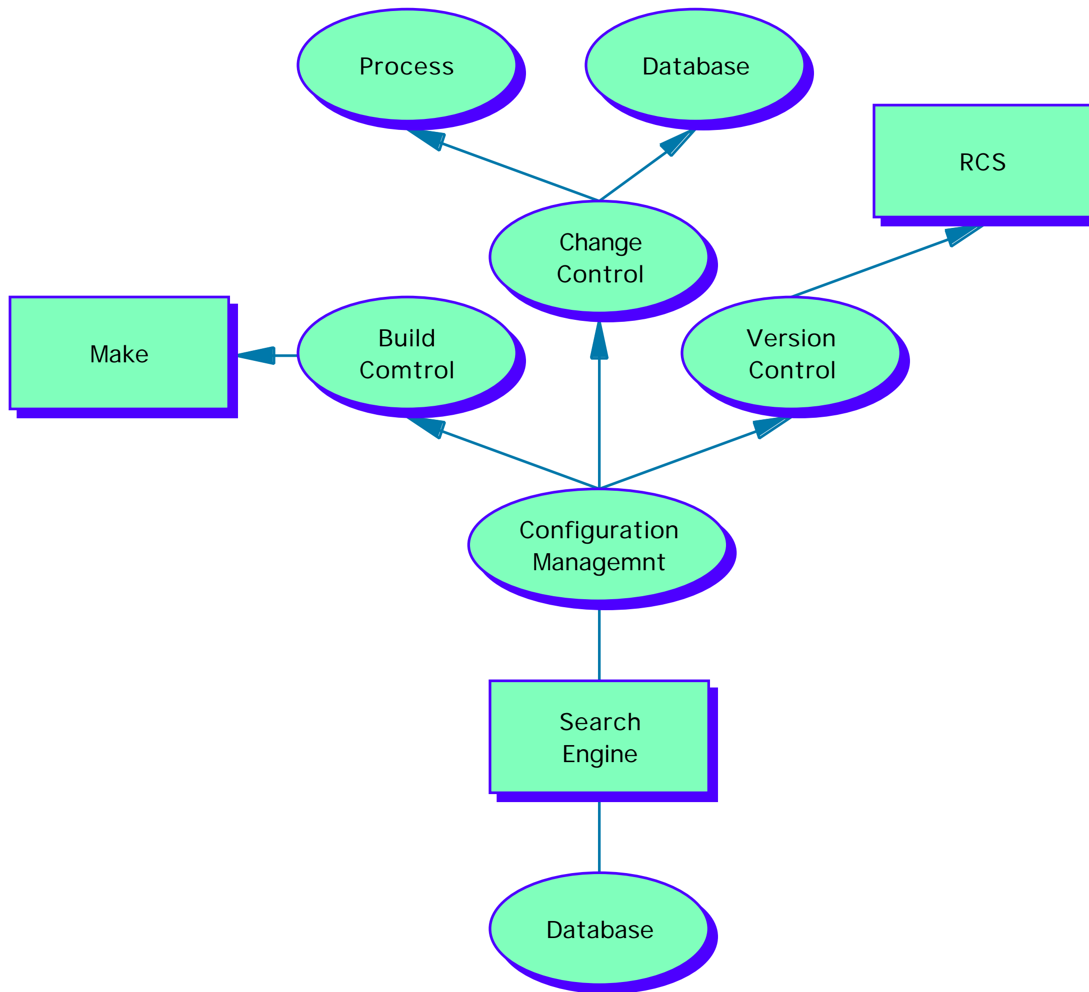
The next step is to classify, or assign a representation, to the assets. Good representations are needed to help users find and understand assets in the library. Once classified the assets are stored in the repository. Users will then submit queries to the repository via a search system. Topics that will need to be considered when creating an reuse library include:

- the platforms used to implement the library storage mechanism
- representation methods for the assets
- search interfaces
- version control
- inter-operability of reuse libraries
- the measurement and evaluation of reuse libraries

Integrated Architecture

The figure below shows a high level proposed architecture for a reuse environment that integrates a reuse library tool with tools for version control, change control, and build control. The ovals indicate needed activities, and the boxes available tools for implementation. A system based on this architecture would provide an environment that would support both reuse and traditional software development activities.

An integrated architecture for configuration management of reusable components



Summary

Configuration management is an important part of software engineering. It includes three inter-related activities: version control, change control, and build control. Existing software development environments, such as UNIX, offer tools to support these activities separately, This paper has discussed configuration management issues on terms of reusable components, and has proposed an integrated architecture for combining tool support for configuration management and reuse libraries.

REFERENCES

Carle, R. (1987). Reusable Software Components for Missile Applications. In Tenth Minnowbrook Workshop on Software Reuse, . Blue Mountain Lake, NY:

Frakes, W. B., & Gandel, P. B. (1990). Representing Reusable Software. *Information and Software Technology*, 32(10), 653-664.

Frakes, W. B., Fox, C. J., & Nejmeh, B. A. (1991). *Software Engineering in the UNIX/C Environment.* Englewood Cliffs, NJ: Prentice-Hall.

Frakes, W., Prieto-Diaz, R., & Fox, C. (1998). DARE: Domain Analysis and Reuse Environment. *Annals of Software Engineering*, 5, 125-151.

Mili, A., Mili, R., & Mittermeir, R. T. (1998). A Survey of Software Reuse Libraries. *Annals of Software Engineering*, 5, 349-414.

Rochkind, M. J., 1975 "The Source Code Control System," *IEEE Transactions on Software Engineering*, SE-1, no. 4 , 255-265.

Tichy, W., "RCS—A System for Version Control," 1985 *Software Practice and Experience*, 15, no. 7 , 637-654.