Sha Liu

Master Thesis in Computer Science

Software Configuration Management and Change Management

Supervisor: Prof. Ivica Crnkovic
Examiner: Prof. Ivica Crnkovic
ABSTRACT

Nowadays, as the use of computers is rapidly spreading to our life, software is getting more and more complex and large in computer systems. Therefore, the software configuration management (SCM) is playing an increasingly important role in the software development process. One of its significant activities is change management, which has an outstanding role in dealing with the continued and concurrent change requirements during the system development and use.

In this report, we describe some basic activities of SCM, overview some representative SCM CASE tools with emphasizing on change management and analyze a possibility of integration of SCM version management tool (e.g., Subversion) and error management tool (e.g., Bugzilla) in order to provide an integrated software configuration management and change management. Moreover, a set of exercises based on RCS are developed, which illustrate some SCM simple activities and their modification to facilitate version management and change management.
# TABLE OF CONTENTS

Part I Overview of Software Configuration Management .......................................................... 1
  1 Introduction .................................................................................................................. 1
    1.1 The Aim of the Thesis .................................................................................... 1
    1.2 What is Software Configuration Management? ............................................. 1
    1.3 What is Change Management? ...................................................................... 1
    1.4 CASE Tools for SCM .................................................................................... 2
    1.5 Overview of Report. ........................................................................................ 2
  2 The Basic SCM Activities ............................................................................................ 2
    2.1 SCM Planning ................................................................................................ 2
      2.1.1 Configuration Items Identification ............................................................ 3
      2.1.2 The Configuration Database ...................................................................... 3
    2.2 Version and Release Management ................................................................. 4
      2.2.1 Revisions and Branches ............................................................................ 4
      2.2.2 Version Identification ................................................................................ 4
      2.2.3 Release Management ................................................................................. 6
    2.3 System Building ............................................................................................. 7
    2.4 Change Management ..................................................................................... 8
      2.4.1 Change Management Procedures and Process .......................................... 8
      2.4.2 Component Derivation History ................................................................. 9
  3 CASE Tools for SCM ................................................................................................ 10
    3.1 The Basic Functions of SCM Tools ............................................................. 10
    3.2 SCM Models ................................................................................................ 10
    3.3 RCS .............................................................................................................. 12
      3.3.1 Introduction ............................................................................................. 12
      3.3.2 RCS Versioning ....................................................................................... 12
      3.3.3 RCS Difference ....................................................................................... 12
      3.3.4 RCS Concurrency and Security ............................................................... 13
      3.3.5 RCS Configuration and Baseline Management ...................................... 13
      3.3.6 RCS Change Management ...................................................................... 13
    3.4 CVS .............................................................................................................. 14
      3.4.1 Introduction ............................................................................................. 14
      3.4.2 CVS Versioning ....................................................................................... 14
      3.4.3 CVS Concurrency and Security .............................................................. 14
      3.4.4 Subcontractor (third party vendor source) Tracking of CVS ................. 14
      3.4.5 CVS Configuration and Baseline Management ...................................... 15
      3.4.6 CVS Change Management ...................................................................... 15
    3.5 Subversion .................................................................................................... 15
      3.5.1 Introduction ............................................................................................. 15
      3.5.2 Subversion Versioning ............................................................................. 16
      3.5.3 Subversion Configuration and Baseline Management ......................... 16
      3.5.4 Subversion Change Management ............................................................ 16
    3.6 Rational ClearCase ....................................................................................... 17
      3.6.1 Introduction ............................................................................................. 17
Part I Overview of Software Configuration Management

1 Introduction

1.1 The Aim of the Thesis

Software Configuration Management (SCM) covers many areas, from SCM planning, to version and release management, to system building and to change management. Moreover, many tools have been developed to facilitate above SCM activities. Some are commercial tools supporting an integrated SCM solution; others are free software focusing on a specific SCM area. However, the most advanced tools are not necessary the best since they can be very complex. In many cases a good combination or modification of simple free tools can achieve the purpose as well. The aim of this thesis is to study software configuration management, overview some representative SCM tools, develop a set of exercises that illustrate some SCM simple activities and their modification to facilitate version management and change management, analyze a possible integration of SCM tools and bug-tracking system in order to provide an integrated software configuration management and change management.

1.2 What is Software Configuration Management?

Current definition would say that software configuration management (SCM) is the control of the evolution of complex systems [2]. More pragmatically, SCM can be defined as the controlled way to manage the development and modification of software systems and products during their entire life cycle [7]. SCM, which is an extension of Configuration Management, more focuses on the software support. Comparing to the other fields in software engineering, the SCM has an outstanding role in dealing with the continued and concurrent change requirements during the system development and use.

Nowadays, the software is getting more and more complex and large in computer systems, it may have many different versions for different hardware, operating systems, fault corrections and also change requirements during its development lifecycle. Therefore, people need to manage evolving systems because it is easy to lose track of what changes have been incorporated into what system version [1]. In contrast, if developers or organizations don’t have efficient SCM, it may waste the massive manpower, time and financial resource in dealing with the history status of the software system and its changes, which may result to the big loss of the market shares and profits. Moreover, as the CMM (Capability Maturity Model) and CMMI (Capability Maturity Model Integration) standards have been widely accepted by many organizations for their measurements and improvements of software development processes, the use of SCM is efficiently facilitated and encouraged as a consequence. Many SCM related standards have been introduced and proposed by the organizations like ISO and IEEE. An example of SCM standard is IEEE 828-1998, which is a standard for SCM planning.

1.3 What is Change Management?

There are many activities within SCM. Typically, four fundamental SCM activities are elaborated frequently. They are SCM planning, version and release management, system building and change management. SCM planning describes the discipline that used for software configuration management. Version and release management are concerned with controlling and tracking on the system versions. System building is the process of assembling software components together according to some predefined build instructions. And change management is the management of
system change.

Change is the essential fact in a large, complex software system. For a large and complex software system, there may exist thousands lines of source codes, test units, specification and design documents during its development lifecycle. Traditionally, ‘waterfall’ model is widely used for the large software development process. Thus, most of the SCM activities are elaborated according to the ‘waterfall’ model for the software development. However, nowadays, as the modern software development approaches based on incremental specification and development for small software system is getting more and more popular (e.g., Agile software development [3], Scrum software development [4], and Extreme programming (XP) development [5]), some specific SCM activities and tools are increasingly used in agile and other rapid software development processes as well. For example, by using some version management and system building tools which can efficiently keep track of the system changes and also manage the system versions.

1.4 CASE Tools for SCM

As the SCM is playing an increasingly important role in the software development process, many tools (e.g., RCS [8], CVS [9], Subversion [15], Rational ClearCase [22], CM Synergy [18], Microsoft Visual SourceSafe [24]) have been produced to support SCM in different areas. RCS, CVS and Subversion are representations for the version management, Rational ClearCase is an integrated workbench supports all the basic SCM activities, Telelogic Synergy includes a series tools to support specific SCM purpose. Paired with above SCM tools, the bug-tacking software is used to help software developers to manage the bugs they find during the software development process. The most representative tool is Bugzilla [30].

1.5 Overview of Report

This thesis report is organized as follows. Part I is an overview of software configuration management. Sections 2 present the some basic SCM activities. Section 3 and 4 overview some well-known SCM CASE tools and Bugzilla with emphasizing on change management. And we conclude Part I in section 5. In Part II, we describe a set of practical exercises based on RCS that illustrate simple functions and their modification to support change management. (In this report the terms component, configuration item and file are used interchangeably.)

2 The Basic SCM Activities

The basic SCM activities include SCM planning, version and release management, system building and change management. In some other books and materials, the SCM activities may be called or interpreted as other similar terminologies, such as SCM functions. In fact, they are more or less the same. Therefore, here we follow the same naming convention as shown in [1]. In this section, four basic SCM activities will be described in more details respectively.

2.1 SCM Planning

As we mentioned before, the SCM planning describes the discipline that used for software configuration management. This activity is particular important for the large software system. The concrete SCM plans may include the following sections:
- Define which configuration items need to be managed or controlled (all documents that may be useful for future system evolution should be controlled by the configuration management system [1]).

- Define who will take the responsibility for the SCM.

- Define the SCM discipline that all team members have to follow.

- Define the SCM tools that all team members have to learn to use.

- Define the structure of the configuration database.

- Define other requirements and constraints for specific project when necessary.

2.1.1 Configuration Items Identification

Normally, the configuration items are represented as project requirements or specifications, designs, implementations and test cases. For a large, complex software system, the aforementioned configuration items are massive and easy to lose track. Therefore, a consistent configuration items identification scheme is necessary for the SCM planning. Obviously, the configuration items identification scheme must assign a unique name to all documents under configuration control [1]. In most cases, a hierarchical naming scheme is used for the configuration items identification, which identifies the type of items and also reflects the relationships between different hierarchies. In [1], a hierarchical naming scheme example is shown as below:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE

As we can see from the example, the hierarchical naming scheme is very easy to understand. However, Sommerville [1] identifies that this naming scheme significantly reduces the opportunities for reuse. He claims that “it can be very hard to find related components (e.g., all components developed by the same programmer) where the relationship is not reflected in the item-naming scheme” [1].

2.1.2 The Configuration Database

The configuration database is a database, which is used to record the controlled configuration items. It is the main part of the SCM planning process. A configuration database does not just include information about configuration items, it may also record information about users of components, system customers, execution platforms, proposed changes and so forth [1]. It is a kind of information database, from where you can get all needed information related to the system configurations (e.g., the latest software version number, the difference between different software versions, who initiates the changes and also who is the responsible person for the changes).

Ideally, Sommerville [1] declares that the configuration database should be integrated with the version management system that is used to store and manage the formal project documents (For the version management systems, we will discuss some representative CASE tools in section 3). However, very few companies follow the above rules due to the finance issues. In most cases, they separate their configuration database from their version control systems and update the configuration database manually. Apparently, this approach may lead to the inconsistency
problem between the configuration database update and the real status of the system.

2.2 Version and Release Management

Version and release management are concerned with controlling and tracking on the system versions. Nowadays, the version management is widely used in the software system development. Many SMC CASE tools support this functionality. Some well-known version management CASE tools like RCS, CVS and Subversion will be illustrated in section 3. Version management is of great significance to the retrieve of the system versions. A system version is an instance of a system that differs, in some way, from other instances [1]. This difference of versions can be the new functionalities, different hardware or software configurations and specific performance based on the additional customer requirements.

A system release is a version that is distributed to customers [1]. Versions of a system have a bigger category than releases. Before a version is finally release to customers, it may have many other versions for different uses (e.g., testing, fix the software bugs).

2.2.1 Revisions and Branches

The revisions here mean the versions of a set of components (configuration items), which are evolved from each other and organized in a sequence. Another terminology frequently used in version management is branches, which organize the versions of components in parallel. These are shown in figure 1. The new branches should be created only at special development stages and conditions. Feiler [6] describes four situations when branches are needed:

- Temporary fixes: when the customer request a fix of a previous release revision.
- Distributed development and customer modifications: when several distributed customer sites operate the same revision for modifications.
- Parallel development: when designate a possible different implementation technique approach for the main design, which can be either adopted or abandoned later.
- Conflicting updates: when two programmers modify the same revision simultaneously.

2.2.2 Version Identification

Version identification is the methodology to identify system version. It is used to identify the versions of system components that construct the system. During a large software system development, there may exist hundreds of software components with different versions. Therefore, a good version identification methodology can easily help to find the required component and significantly reduce the ambiguity and misunderstanding between different component versions. However, the previously discussed configuration items identification scheme can not apply here, since there may be more than one versions of each configuration item.

Version identification has been a major research topic in SCM. Nowadays, three version identification schemes are widely used and accepted in version management. We will discuss them in the following sub-sections one by one.
2.2.2.1 Version Numbering

Version numbering is the most commonly used version identification scheme by adding an explicit and unique version number to the component or system name. Figure 1 helps to illustrate how it works. The letter V represents the component or system name. If the first release version is V1.0, then the subsequence versions numbers will linear increase until the new release version is created (e.g., V2.0). In principle, any existing version may be used as the starting point for a new version of the system [1]. That explains why in figure 1, version 2.2 is created from version 1.2 rather than from version 2.1.

![Figure 1: Version numbering scheme](image)

The version numbering scheme is easy to understand and simple to use. However, comparing to the other version identification schemes, it reveals nothing about the component attribute information (e.g., development language, status and platform) or the relationships between changes and versions. Only those experienced people, who are the developer of the components or who are familiar with component development can realize the difference of versions. Therefore, it is necessary to complement this kind information of versions in the configuration database consequently.

2.2.2.2 Attribute-based Identification

As we discussed previously on version numbering scheme, it reveals nothing about the component attribute information (e.g., development language, status and platform). Therefore, the attribute-based identification scheme is proposed in order to complement this shortcoming by identifying the version as a unique set of attributes. For example, the version of the software system AC3D developed in Java for Windows XP in January 2003 would be identified: AC3D (language = Java, platform = XP, date = Jan2003) [1]. As a consequence, people can search and retrieve the specific version by specifying its attribute.

In some respect, the attribute-based identification and the version numbering can be associated together. For example, the attribute-based identification system may be built as a layer on top of a hidden version-numbering scheme, the configuration database then maintains the links between identifying attributes and underlying system and component versions [1].

2.2.2.3 Change-oriented Identification

Although the attribute-based identification scheme complements the component attribute information problem of the version numbering scheme, it still do not solve the problem about
relationships between changes and versions. Additional works still need to be implemented in the configuration database that describes the change relationship of each version.

The general principle of the change-oriented identification is to link an association between the system change and its required change set of the system components. One thing need to be pointed here is that change-oriented identification is used to identify system versions rather than components [1]. Ideally, the required change set of the system components can be implemented randomly or systematically. However, in practice, it doesn't work like that. Sometimes, the change set is incompatible and conflicts with each other.

Since the change-oriented identification can be seen as the complementary of the traditional version numbering scheme in some respects, the two schemes can be combined together to represent the system versions as well. This can be illustrated in figure 2 [6]. The shaded box (Base) is the baseline of system versions. Whereas, the shaded ellipses (CS1, CS2 and CS3) are the change sets. The new system release R2.0 can be expressed in terms of the baseline (Base) and the change sets (CS3) in addition.

![Figure 2: The combination of version numbering and change-oriented identification](image)

### 2.2.3 Release Management

The system release management is responsible for deciding when the system can be released to customers, managing the process of creating the release and the distribution media, and documenting the release to ensure that it may be re-created exactly as distributed if this is necessary [1]. Thus, the possible release assets can be whatever needed for the release (e.g., configuration files, data files, documentations). One important thing in release management is that the new release version has no dependency on the installation of previously releases, since it is not obligatory for customer to update the latest version of the system.

#### 2.2.3.1 Release Occasion

Choosing the right occasion of the system release is one of the important parts in the release management. Sommerville [1] summarizes some typical factors may influence system release:

- For serious system faults.
- For different operating system platforms.
- For significant new functionality.
- For fierce competitions.
• For marketing requirements.
• For specific customer requirements.

2.2.3.2 Release Creation and Documentation

The release creation is concerning on the creation of all the needed release assets based on the release directory, which is handed over for customer distribution. The release documentation is somewhat like the “archives” for the company use within intra-organization. These kind “archives” may include the source code versions, executable code versions, configuration files, operating systems, compilers, libraries and all the other integrant items used to ensure the recreation system is exactly the same as before whenever it is needed.

2.3 System Building

The system building (or called build management) is the process of assembling software components together according to some predefined building instructions. Before building the system, there are a number of pre-requirements checklists. For example,

• Whether the predefined building instructions include all the required software components?
• Whether the software components version is appropriate?
• Whether the required data files are available?
• Whether the referenced data file name or the route of directory is consistent with the one on the target machine?
• Consider other requirements and constraints for specific project when necessary.

Nowadays, software configuration management tools or, sometimes, the programming environment (such as Java development environments, the build script is created automatically by parsing the source code and discovering which components are called) are used to automate the system-building process [1]. Figure 3 [1] shows the process of the SCM tools support for the system building. Firstly, a build script is created to define the components dependencies and the system-building tools used to compile and link the software components. Secondly, according to the predefined build script, the system builder will find all the required software components with appropriate versions by using version management system. Thirdly, the system-building tools decide when the source code of components must be recompiled and when existing object code can be reused. Finally, the executable system is built from its components. For aforementioned approach, a potential problem may occur if the version management and system-building tools are not integrated together. That is when there are multiple source code files representing multiple versions of components, it may difficult to tell which source files were used to derive object-code components [1]. This confusion is particularly likely when the correspondence between source and object code files relies on them having the same name but a different suffix (e.g., .c and .o) [1].
2.4 Change Management

Change is a fact of life for large software systems [1]. Therefore, change management is particularly important and necessary to ensure the software changes in a controlled way. Change management is the management of system change. In [2], the author mentions that they had made a survey about “what are the most useful features of SCM” at the SCM9 conference (September 99 in Toulouse France). The result shows that most of the interviewees agree on that the change control is the most useful and appreciated feature. The author also indicates that this answer is pretty much consistent among all the interviewees, no matter where they come from and which SCM tools they use.

Normally, there are two main objectives of change management. The first objective is to provide the change management procedures and process. Such processing includes change identification, analysis, prioritizing, planning for their implementation, decisions for their rejection or postponement, and decisions for their integration in new product releases [7]. The second objective is to track component derivation history.

2.4.1 Change Management Procedures and Process

Within a company, especially the well-organized big company, they set their own change management procedures and process based on the SCM standards and also incorporated into their quality handbook. Change management procedures are concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components of system have been changed [1]. It is unnecessary for all the changes to the system to follow the aforementioned procedures, only for the changes to the controlled system. The controlled system is the software system which is approved by the quality assurance (QA) team for some acceptable quality. Therefore, any changes to the controlled system need to be analyzed and approved by the relative responsible engineers and departments. Controlled systems in some other materials are called baselines as well, since it is the beginning for the further controlled development and evolution.

The change management process includes several stages. When someone wants to initiate a change, he or she needs to fill the change request form (CRF) firstly, see figure 4. Normally, the CRF is used to record the details of the change request, like the reason why need to change and how to change. However, the real CRF format can be different according to the definition of different companies or the SCM planning for specific projects. For small project by using agile development, this process may not be applicable. Being that it may slow down the development process, which is the essential factor of agile like rapid development approaches. After completing the CRF, the next stage is to register it in the configuration database. Later on, the change request is analyzed its validation by the relative responsible committee within some predefined period time. The committee is composed mainly of relative engineers, experts and
other responsible people, who have the authority to validate the change requests. The change request can only be valid when all the members in the committee agree on that change. Otherwise, it is rejected. Of course, the committee should give the reason why it has been rejected at last.

<table>
<thead>
<tr>
<th>Change Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity: CR2</td>
</tr>
<tr>
<td>Title: Implement change management by adding change request</td>
</tr>
<tr>
<td>Change requester: Sha Liu</td>
</tr>
<tr>
<td>Initiate Date: 29/05/2008</td>
</tr>
<tr>
<td>Due Date: 09/06/2008</td>
</tr>
<tr>
<td>Priority: High</td>
</tr>
<tr>
<td>Description: When check in a file, refer to a change request and automatically update it. Implement change management by adding change request. We use the RCS open source code for windows by adding -c option of check in, e.g., ci -cChangeRequest file</td>
</tr>
<tr>
<td>Components changed:</td>
</tr>
<tr>
<td>test.txt</td>
</tr>
<tr>
<td>XXXXXXXX2.txt</td>
</tr>
<tr>
<td>X3.c</td>
</tr>
</tbody>
</table>

Figure 4: A sample change request form

Nowadays, as the use of internet is rapidly spreading to our life, the change management process turns to web-based development. When a change request is submitted on-line, a notification will automatically send to the relative committee members to remind them to go through it.

For valid changes, the next stage of the change management process is the cost analysis of making the change, including the actual cost and also the potential cost by taking into account some change scenarios. A change control board (CCB) is responsible to justify and approve changes from an organizational strategic or economical view out of a technical point of view. The reason why the CCB need to present is obviously simple, since companies need to significantly reduce the develop time and cost for the new software due to the fierce competitions. Therefore, although some change request has been validated, it can still not be approved finally.

2.4.2 Component Derivation History

Component derivation history means to record and maintain the software components change history. A good way to keep the derivation history is in a standardized comment at the beginning of the component source code [1]. Figure 5 shows a component header information example elaborated in [1]. As we can see from figure 5, it is a good custom for the developer or the programmer to write the component source code in a standard clear way. However, the typical organizational scope of this kind approach is intra-organization.

```plaintext
// BANKSEC project (IST 6087)
// BANKSEC-TOOLS/AUTH/RBAC/USER_ROLE
// Object: currentRole
// Author: N. Perwaiz
// Creation date: 10th November 2002
//
// (c) Lancaster University 2002
//
// Modification history
// Version  Modifier  Date        Change          Reason
// 1.0     J. Jones  1/12/2002  Add header     Submitted to CM
// 1.1     N. Perwaiz 9/4/2003  New field     Change req. R07/02
```

Figure 5: Component header information
3 CASE Tools for SCM

In the development of human historical experience, sophisticated suitable tools can facilitate the work efficiently. Thus, SCM tools are necessarily used to support developers for their SCM works. Nowadays, a lot of SCM tools are available for that purpose. Some of them are commercial tools (e.g., Rational ClearCase [22] is an integrated workbench supports all the basic SCM activities), and others are free open-source tools (e.g., RCS [8], CVS [9] and Subversion [15] support for the version management). In this section, we first describe the basic functions supported by SCM tools. Then, we talk about common CM (configuration management) models adopted by current SCM tools, and follow with the descriptions of some of typical SCM tools’ characteristics in details.

3.1 The Basic Functions of SCM Tools

Some basic functions provided by SCM tool are:

- Versioning: store, control, and track the components (configuration items) development history.
- Check out: retrieve the components (configuration items) from the repository for read or write, since the components can not be directly accessible in the repository.
- Check in: store the modified components (configuration items) back to the repository, thus a new version of the components will be created.
- Baseline: is the beginning for the further controlled development and evolution.
- Difference (deltas): the difference between versions are managed by using delta algorithms, which save the disk space efficiently.
- Merge: combination of two versions into a new version.
- Configuration: is a set of revisions. A useful technique for the specification of a configuration supported by several systems is to offer a rule-based selection mechanism [7].
- Concurrency: support or prevent concurrent access by several developers.
- Distribution: support the distributed global work by developers.

3.2 SCM Models

Feiler [6] identifies four CM models adopted by SCM tools. However, most of current commercial CM systems implement the first three models. They are:

- The check-in/check-out model.
- The composition model.
- The long transaction model.
The change set model.

The check-in/check-out model provides version support of individual system components (configuration items). This model can be found in the earlier SCM tools like RCS [8], which consists of two tools: a repository tool and a build tool. The repository tool stores the versioned components and controls the creation of new versions. It offers versioning maintenance and concurrent control mechanisms by using version branching as well as version merging and branch locking. If a developer wants to modify a specific versioned component, first he/she needs to check out the component from the repository to his/her working space, then the component is locked in the repository, which means that no other developers can check out this component until it is check in to the repository within a new version and the lock is released. The build tool, given a description of the components that make up a product, automates the generation of derived files, such as object code and linked executables, through application of tools to source files [6].

The composition model, which consists of a system model and version selection rules, relies on the check-in/check-out model and improves support on system configurations. The system model reveals the components composition of the system. In details, it provides a link among configurations, system build, and language systems. This link permits the CM system supporting the composition model to include management of derived objects and checking of interfaces between components as well as between aggregates, i.e., subsystems [6]. Whereas, the version selection rules indicate the exact versions of the components used to construct the system, which allow the user to identify the version selection rules in two ways: by searching on labeled version (e.g., indicate the exact component version number) and by predicating on version attributes (e.g., make a query to select the latest version). The typical SCM tools which uses the composition model is Apollo DSEE [23].

The long transaction model manages the system configurations as a series of small changes and coordinates these changes by concurrent team activities. Comparing to the previously mentioned two models, the long transaction model which associates with the system configurations rather than the individual components. It provides stable workspaces, in which the developer is isolated from external changes, and a concurrency control scheme which is used to facilitate the coordination of concurrent changes. The represented SCM tools using long transaction model is Sun NSE [25].

The change set model supports configuration management of logical changes which are made up by a set of components modifications. The change set model consists of a baseline and a set of change sets. Users can associate with change sets directly. Comparing to the other three CM models, the change set model focuses on the change-oriented configuration management rather than the version-oriented configuration management. The change set concept provides a natural link to change requests, which contain the detailed change set information. Typical change set concept are patches, which are the set of differences (delta) between configuration versions (see figure 6 [6]). Typically, CM systems which implement the change set model usually integrate with the check-in/check-out model, since the change set model does not provide concurrency control (e.g., Software Maintenance & Development Systems Aide-De-Camp (ADC) [26]).
3.3 RCS

3.3.1 Introduction

The Revision Control System (RCS) [8] is an early free SCM tool that provides version control. It automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations [8]. Originally, RCS is designed for program-oriented version control. However, it is applicable for any text documents management, such as documentation, source code, test data and etc.

RCS contains basic SCM functions like versioning, check-in/out, baselines, difference, merge, configuration and concurrency. More pragmatically, RCS manages the revisions by an ancestral tree, which the initial revision is represented as the root of the tree and the tree edges show the evolutions between revisions. RCS can automatically update the versioning and identification of components by ‘stamping’ with the marker $Id$ when required. It also offers the merge function to combine two modified components versions from different branches into a new version in one of the branches. Baselines can be checked out by setting a label on specific revisions of the components. Besides revisions management, it provides several mechanisms for the storage of the differences (deltas) between versions in order to save the space. Moreover, it provides flexible selection functions for composing configurations. However, the main problem for RCS is that it is not suitable for the concurrent access by several developers as implementing the locking mechanism. All of aforementioned concepts will be discussed in details in the following sections.

3.3.2 RCS Versioning

RCS deals with revisions in a file-oriented way, all the versioned files are under a single directory. In details, RCS manages the revisions by an ancestral tree, which numbers the initial revision as 1.1, increases the numbers (e.g., 1.2, 1.3) in sequence for its successive revisions and in parallel (e.g., 2.1, 3.1) for its branches.

3.3.3 RCS Difference

RCS provides several mechanisms for the storage of the differences (deltas) between versions in order to save the space. The deltas are employed in RCS by using line-based scheme rather than the character-based scheme, since the latter is too much time consuming. However, using deltas is a classical space-time tradeoff: deltas reduce the space consumed, but increase access time [6]. For RCS, it stores the most recent revision of the components as a completely version. All other revisions are stored as reverse deltas, which are used to retrieve a specific preceding revision. This enables a fast check in/out the components to/from the repository using a copy operation. For check in, the previous revision is replaced by a reverse delta. If the component version has the branches, the forward deltas are used for branches from the transition points.
3.3.4 RCS Concurrency and Security

As we talked before, RCS adopts the check-in/check-out model for system implementation. Therefore, it offers concurrent control by using locking mechanism, which prevents concurrent access by several developers. For simple explanation, the locking mechanism can be interpreted as a “Lock-Modify-Unlock” scenario. When a developer checks out the file for modification, the component is locked in the repository automatically. This locked component will not be released until the modified component check in to the repository. The simple locking mechanism maximally prevents the occurrence of the modification overriding. Thus, RCS is particular for small group activities within atomic updates. Moreover, RCS facilitates the security control by using an access list, in which specifies the access authority and priority for the people who want to modify components.

3.3.5 RCS Configuration and Baseline Management

A configuration is a set of revisions, where each revision comes from a different revision group, and the revisions are selected according to a certain criterion [6]. RCS provides many selection functions for configuration management:

- Default selection: retrieves the latest revision of components.
- Release based selection: retrieves the latest revision of a release.
- State and author based selection: retrieves the revision of components by indicating their versioning and identification (e.g., state attributes or author names).
- Date based selection: retrieves the revision of components by indicating their identification (e.g., release date).
- Name based selection: retrieves the revisions of components by assigning symbolic names, which significantly simplifies the check out process when many components in a large system need to be checked out once. RCS facilitates frozen of a configuration for a large system with many components.

Above bulleted items can be classified to two categories: the partially bound configuration and bound configuration. Obviously, the first two bulleted items are partially bound configurations, since the result can be different in time. However, the last three bulleted items are bound configuration, which can be used to form a baseline as a consequence.

3.3.6 RCS Change Management

Since RCS is designed to support the version management originally, we develop a set of exercises which improve the RCS ci command performance by adding -e option to support the change management. That is: when check in a component (file), refer to its associated change request (CR) and automatically update it. By this way, we can easily track the component derivation history within associated CRs. For more details, please refer to section 5.7.
3.4 CVS

3.4.1 Introduction

The Concurrent Version System (CVS) [9] is a RCS based free SCM tool that supports version and release control. In contrast with RCS, one of the powerful features of CVS is to allow several developers to access the same components (files) concurrently. Moreover, CVS is one of the few SCM tools that provide the functionality of tracking the subcontractor (third party vendor source). The extensive usage of shareware in many software implementation projects has increased the importance of such control mechanism [12].

Since CVS is built upon the RCS, it contains RCS functions like versioning, check-in/out, baselines, difference, merge, configuration and concurrency. However, comparing to RCS, CVS extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories each containing revision controlled files [9]. Moreover, CVS supports distribution via LAN or Internet. CVS uses a client-server architecture that the versioned files are stored in a central server repository and users check out a copy of all the files in distributed client machines. It facilitates the concurrent modification from multi-developers by using conflict-resolution algorithms. Baselines or software release can be checked out at any time by setting a symbolically tag on specific revisions of the components. Moreover, it provides flexible user configurations, such as module database, configurable logging support, tagged releases and dates. All of aforementioned concepts will be discussed in more details in the following sections.

3.4.2 CVS Versioning

In contrast with RCS, CVS deals with revisions in a directory-oriented way. That is: CVS basic version control functionality maintains a history of all changes made to each directory tree it manages, operating on entire directory trees, not just single files [13]. However, the revision numbers are still per-file in CVS, since CVS uses RCS as a basis.

3.4.3 CVS Concurrency and Security

CVS provides conflict-resolution algorithms to facilitate the concurrent modification of the same files by multi-developers. For simple explanation, the conflict-resolution algorithms can be interpreted as a “Copy-Modify-Merge” scenario when applying the concurrent access. First, different developers can check out the same file independently, which allows the isolation work among multi-developers. Then, before the user’s check out file to be committed as a permanent change, CVS uses the “update” command to merge the difference between the user’s modification and those modifications which have been committed early by other developers. If any conflicts occur during the merge, CVS will let the user to resolve the conflicts consequently. In contract with RCS, this conflict-resolution approach significantly improves the productivity and efficiency of the software development by supporting multiple developers’ access simultaneously. For CVS security control, Berliner [9] indicates that there is no significant mechanism for the security of the source repository, which only adopts the usual UNIX approach at that time. However, as the increasing development of CVS, the remote operation can be authenticated by using the industry standard protocols to verify the identities of the remote users (e.g., rlogin, rsh, ssh).

3.4.4 Subcontractor (third party vendor source) Tracking of CVS

One of the outstanding features of CVS is to provide the functionality of tracking the subcontractor (third party vendor source). This feature is particular important for current software
development, since many software are based on the source distribution from the subcontractor. However, it is often needed to modify the software in order to adapt these new distributions. Thus, the tracking of future subcontractor’s release is necessary. CVS implements this task by preserving the directory hierarchy of the subcontractor’s distribution when initially set up a source repository. When a new version of subcontractor releases, CVS adds the new release version to the existing preserved subcontractor’s directory correspondingly. As a consequence, CVS will remind the developers to merge the new subcontractor release together with their modifications. However, if the vendor changes the directory structure or the file names within the source distribution, CVS has no way of matching the old release with the new one [9].

3.4.5 CVS Configuration and Baseline Management

CVS provides flexible user configuration management. Some major representations are module database, configurable logging support, tagged releases and dates. Similar to the frozen of a configuration by RCS, the module database can assign symbolic names to collections of directories and files for the large system retrieve. Moreover, the database records the physical location of the sources as a form of information hiding, allowing one to check out whole directory hierarchies or individual files without regard for their actual location within the global source distribution [9]. The configurable logging support is aiming to save the log messages by using an arbitrary program (e.g., file, notesfile, database, or email) when the check out files commit to the repository. This efficiently helps to track the changes history of the software development. Tagged releases and dates mean that user can retrieve the releases and files that have been tagged in the repository by indicating their tag names and tag dates.

3.4.6 CVS Change Management

Like RCS, CVS is originally introduced to support the version management as well. Although we can audit log of what is changed by CVS to track component derivation history, it still can not be regarded as change management tool.

3.5 Subversion

3.5.1 Introduction

Subversion [15] is a free SCM tools supporting version control. It is initially designed as a functional replacement for CVS. That is, Subversion aims to produce a compelling replacement for the CVS [14]. Therefore, Subversion adopts most of CVS features and enhances the performances by fixing some obvious CVS flaws. Below is a list of some Subversion significant features:

- Directories as well as files are versioned: This allows copying, moving, renaming and other rearrangements of files.

- Support the handling of binary files and fast efficient network access: Subversion can efficiently handle binary files as well as text files by using a binary differencing algorithm. Moreover, this algorithm is used to transmit the deltas in both directions over the network (CVS transmits deltas only from server to client). Thus, the costs of this transition are only proportional to deltas size, not to that modified file size.

- Make truly atomic commit: Comparing to CVS, Subversion makes a commit entirely, no partial commit allowed. The log messages are attached to the
revision per commit, which sufficiently decreases the log message redundancies.

- Support the concept of changesets: Subversion facilities the changesets (a set of modifications) within an atomic commit. For example, in your working copy, you can change files’ contents, create, delete, rename and copy files and directories, and then commit the complete set of changes as a unit [15].

- Apache Web Server: Subversion can use the HTTP-based WebDAV/DeltaV protocol for network communications, and the Apache web server to provide repository-side network service [13]. This allows Subversion to use many features provides by Apache for free (e.g., authentication, wire compression, and basic repository browsing).

3.5.2 Subversion Versioning

Subversion handles versioning of directories as well as files. It manages the versioned components by a filesystem tree, which is a typical hierarchy of files and directories. Each commit results in an entirely new filesystem tree, and each of these trees is labeled with a single global revision number sequentially (see figure 7 [15]). As we can see from figure 7, the initial revision is numbered as zero and this number will be continuously increased after each successive commit. In contrast with CVS, different revision numbers of a file in Subversion do not necessarily mean any solid modifications, since Subversion tracks the tree structure by versioning the filesystem tree while CVS tracks the file content by versioning the file. Thus, the revision number of a Subversion file does not necessarily shows its evolutionary history.

![Figure 7: Subversion filesystem tree](image)

3.5.3 Subversion Configuration and Baseline Management

Subversion provides flexible user configuration management including many optional behaviors. In contrast with CVS, it supports arbitrary metadata (properties) to be versioned as well as files and directories. That is: Subversion allows adding, moving, modifying and other rearrangements of versioned properties of the directories and files. These properties can be thought of as an invisible set of two-column tables that map property names to arbitrary values attached to each file and directory. You can invent and store any arbitrary key/value pairs you wish: owner, perms, icons, app-owner, MIME type, personal notes, etc [14]. Moreover, in addition to versioning properties, Subversion can version the symbolic links for UNIX users.

3.5.4 Subversion Change Management

Subversion is initiated to edit imperfections of CVS. Quickly, it becomes a widely used version
management tool. As a version management tool, it handles the changes within components. However, for a completed change management, integration with relative change and error management tools is needed (e.g., Rational ClearQuest [22], Telelogic Synergy [18] and Bugzilla [30]). In section 4, we will analyze the possible integration between Subversion and Bugzilla in more details.

3.6 Rational ClearCase

3.6.1 Introduction

Rational ClearCase [22] is a commercial SCM tool supporting all the basic SCM activities, from version management, to change management, to system building. Mostly, it is used for large and medium scale projects within many developers. Some included tools are:

- Rational ClearCase: supports all the basic SCM activities.
- Rational ClearCase MultiSite: enables parallel development across geographically distributed teams.
- Rational ClearQuest: is a change and defect tracking system.

Rational ClearCase provides basic SCM functions like versioning, check-in/out, baselines, difference, merge, configuration, concurrency and distribution. It can handle large numbers of binary files as well as text files. Moreover, it supports parallel development, which enables multiple developers to work simultaneously. For the storage of the differences (deltas) between versions, ClearCase uses interleaved deltas rather than the traditional reverse deltas and forward deltas, where all deltas are written to a single structured file. For distribution, ClearCase, like many other SCM tools, provides replication of data in many distributed servers. However, it has its MultiSite to synchronizing these distributed servers.

3.6.2 ClearCase Versioned Object Base and Views

A versioned object base (VOB) is a repository that stores versioned configuration items. However, ClearCase VOB is different from the typical SCM tools repository, since it uses the file system notion to represent the stored configuration items. That is, ClearCase VOB displays their content as files residing in a file system [20]. Another distinct feature of ClearCase is that the users access to specific versioned configuration items in a VOB by using a view, which is represented as a directory. Views can efficiently facilitate the selection of a configuration of a system. There are two types of views in ClearCase:

- A snapshot view: makes local copies of the configuration items from the VOB to the user’s workspace.
- A dynamic view: enables the direct access configuration items in the VOB.

3.6.3 ClearCase Versioning

ClearCase handles versioning of directories. All files under a directory, including the directory are versioned. However, comparing to the Subversion, the version numbers of the files and directories in ClearCase can be different. Initially, there is only one main branch for each configuration item, which represents the major milestones or released versions of its development. However, as the development of the configuration item, it may have multiple subbranches (named
as branch-types) for a separate development, such as the new functionality implementation or bug fix, which can be merged to the main or to the other branch-types later. Figure 8 [22] shows a typical ClearCase configuration item branch evolution. The ClearCase branch-type efficiently facilitates the concurrent modification of the same file by multi-developers, since a file can be checked out by several different branch-types at the same time. However, synchronization within a branch-type is by locking, preventing developers working on the same branch-type to simultaneously change the same file [27].

3.6.4 ClearCase Configuration and Baseline Management

ClearCase manages the configuration in the form of a configuration specification (config spec), which provides flexible rules used by a view to select versions of configuration items. Each view has an associated configuration specification which lists rules for selecting a version of each needed file [27]. Users can specify the rules by using various attributes and identifications to indicate which configuration items versions to select. The rules can be partially bounded, allowing the user to query the latest version of files, or it can be bounded by indicating the versions of files or their labels. For the latter case, views can be represented as a version of a configuration or a baseline consequently.

3.6.5 Rational’s Unified Change Management

In contrast with aforementioned basic SCM capabilities, Rational ClearCase initiates activity-based SCM capabilities, known as Rational’s Unified Change Management (UCM). UCM defines an activity-based process for managing changes that teams can apply in their development projects, from the specification of requirements and design through implementation and delivery [7]. The activity can be a new function or bug fix, which is automatically associated with a set of files, i.e., change set. This can efficiently help to identify which activities and files are included in each baseline or release.

3.6.6 Rational ClearQuest

Rational ClearQuest is an activity-based change and defect tracking system. Together with UCM, it can provide for a proven change management process. ClearQuest can manage all types of change requests, including defects, enhancements, issues, and documentation changes with a flexible workflow process, which can be tailored to the organization’s specific needs and the various phases of the development process [22].

In [22], some advanced features of ClearQuest are emphasized:
- Customized defect and change request fields.
- Advanced user interface, queries, chart and report capabilities.
- Out-of-the-box predefined configurations.
- Automatic e-mail notification and submission.
- Email, XML, console, and HTTP APIs.
- Available integration with ClearCase to support all SCM activities.
- Available integration with UCM to enhance change management process.
- Deployment independent (Windows, Linux, UNIX and Web).

3.7 Synergy

3.7.1 Introduction

Telelogic Synergy [18] is a powerful family of tools for task-based change and configuration management (CM), which accelerates the release management and builds management processes, maximizes the efficiency of limited development resources, and unites distributed development teams [19]. It integrates with a central database, which facilitates the distributed teams to have a better and confident way to communicate and collaborate during the software development. Some typical tools are:

- Synergy/CM: is a task-based configuration management software, which provides a flexible and powerful repository, and supports a team-oriented workflow approach to software development.
- Synergy/Change: is a web-based change request tracking and reporting system.
- Synergy/CM Distributed Change Management: supports for remote and distributed change management.
- Synergy/CM ObjecMake: is a system building software.

Synergy supports basic SCM functions like versioning, check-in/out, baselines, difference, merge, configuration, concurrency and distribution.

3.7.2 Synergy/Change

Synergy/Change is a web-based, change management solution for request tracking and reporting that easily integrates with existing development and application lifecycle management (ALM) environments [34]. It can provide interfaces to support change management in Telelogic Synergy, IBM Rational ClearCase and Subversion. With Synergy/Change, you can link change requests to the implementation tasks, create configurations based on the completed change requests and generate top-down and bottom-up reports etc.

In [34], some advanced features of Synergy/Change are emphasized:
A User-friendly intuitive change request interface.

Advanced user interface, queries, chart and report capabilities.

Out-of-the-box templates, processes and procedures from professional change management.

Available integration with leading SCM tools (Telelogic Synergy, IBM Rational ClearCase and Subversion).

Web-based interface allowing for distributed teams.

4 Change and Error Management Tool: Bugzilla

4.1 Introduction

Change management and error management are two highly related development activities. Bugzilla [30] is a well-known, open-source bug-tracking software. The bug-tracking software is used to help software developers to manage the bugs they find during the software development process, especially during the testing and deployment phases. For many years, when people work on the bugs of software projects, it is popular to list them on a shared spreadsheet. However, as the bug capability increasing, more and more people need to access and input data on it, and then problem may occur. Moreover, use a public defect-tracker (e.g., Bugzilla) to describe the defects instead of the traditional local defect-tracker running on personal computer, which can efficiently facilitate the accessibility and resolution of defects and reduce the administrative overhead of managing local defect-tracker. It can help to overcome the security access problem between organizations as well. Although some commercial SCM tools (e.g., IBM Rational ClearCase and Telelogic Synergy) support bug-tracking activities as well as other completed SCM activities, they are not applicable for the small scale project or those mainly focus on the bug-tracking applications. Therefore, Bugzilla quickly becomes a popular bug-tracking software due to its free of charge and well development to various situations. Known uses currently include IT support queues, Systems Administration deployment management, chip design and development problem tracking (both pre-and-post fabrication), and software and hardware bug tracking for luminaries such as Redhat, Loki software, Linux–Mandrake, and VA Systems [28]. Nowadays, many big and global companies, organizations use Bugzilla for their projects, such as NASA, Facebook and Nokia. Among them, one need to be emphasized is Mozilla, an open-source browser project. Actually, the Bugzilla originally is built for Mozilla project. However, as time goes by, it has continuously improved and changed a lot to become a general bug-tacking system.

In [28], some advanced features of Bugzilla are emphasized:

- Integrated, product-based granular security schema. This allows the administrator to specify who has the right to modify the bugs.

- Inter–bug dependencies and dependency graphing. Figure 9 shows a sample dependency graph, from which we can see the inter-bug dependencies of bug 315940 clearly.
Advanced reporting capabilities. Bugzilla has many ways to find bugs information. One is reports, which intuitively shows the current state of the bugs, and another is charts, which shows the change over time of the bugs. There are two main categories of the reports, Tabular reports and Graphical reports. The Tabular reports describe bugs as tables of bug counts in 1, 2 or 3 dimensions, as HTML or CSV. Whereas, the Graphical reports describe bugs based on line graphs, bar and pie charts. Once you’ve generated a report, you can switch different view format from HTML to CSV, to Bar, to Line or to Pie, see figure 10. Similar to reports, Bugzilla currently has two charting systems, Old Charts and New Charts. Just as their names implies, Old Charts chart the change over time and New Charts are the future, which allow you to chart anything you can define as a search. Figure 11 shows a sample Old Charts of Firefox fixed bug counts over years.
A robust, stable relational database management system (RDBMS) back-end.

Extensive configurability. Figure 12 shows a Bugzilla screen view of a particular bug. Some Bugzilla concepts are shown on it:

- Status: These define exactly what state the bug is in - from not even being confirmed as a bug, through to being fixed and the fix confirmed by Quality Assurance [29].
- Product and Component: A product may include several or hundreds of components or even more.
- Version: Indicate the version of the bug component.
- Assigned To: The responsible person for the bug fixing.
- Importance: This indicates how important the problem is.
- CC list: An e-mail list of people who want to know the bug’s status at the first time.

The Product, Component, Version, Status, and Reporter fields are related to tracking, whereas the Summary, Status Whiteboard, Keywords, Severity, Attachments, and Dependencies fields are related to fixing it [31]. These fields contain the data you must input when you’re reporting a bug and the data that helps you filter your searches [31].
A very well-understood and well-thought-out natural bug resolution protocol. The resolution field indicates what happened to the bug. Some resolution concepts are:

- FIXED: The bug has been fixed.
- INVALID: The problem described is not a bug.
- WONTFIX: Indicate a bug which will never be fixed.
- DUPLICATE: The problem is a duplicate of an existing bug.
- WORKSFORME: All attempts at reproducing this bug were futile, and reading the code produces no clues as to why the described behavior would occur [30].
- INCOMPLETE: The bug is vaguely described with no steps to reproduce.

- Email, XML, console, and HTTP APIs.

- Available integration with automated software configuration management systems, such as CVS (through the Bugzilla email interface and checkin/checkout scripts). After enabling Bugzilla email integration, ensure that your check-in script sends an email to your Bugzilla e-mail gateway with the subject of “[Bug XXXX]”, and you can have CVS check-in comments append to your Bugzilla bug [29]. However, Makris et al. [35] identifies that this approach is not synchronous. For example, if a user accidentally commits against the wrong bug number, or a bug against which he is not the owner, the SCM system will proceed with the commit action regardless, without giving the user the option to correct his actions [36]. Additionally, if the email gateway is not active, the developer will not be immediately aware that integration failed [35]. Instead, they propose a generic integration of SCM (e.g., CVS, Subversion) and bug-tracking
system (e.g., Bugzilla), named Scmbug [33]. We will talk about it in more details in section 4.2.

- Web interface allowing for distributed teams.

4.2 Integrating of Bugzilla with Subversion

4.2.1 Introduction

Paired with Subversion, Bugzilla is a bug-tracking system. Both of them are highly related to the change management. Although we can audit log of components changes by Subversion (e.g., we can get changelog information between different revisions by using command ‘svn log -r <rev_old>:<rev_new>’ to track component derivation history), it is still uncertain which change request is associated with these components changes. In the mean time, we can exam the defect report by Bugzilla to track why need to change, it is still unknown what changed in component according the defect. Hence, integration of Bugzilla with Subversion can perfectly solve above questions by telling you why need to change and what changed in component.

To integrate Subversion with Bugzilla, currently there is one project, Scmbug [33]. Scmbug provides integration of SCM with bug-tracking system, which helps to solve the integration problem of many dominant SCM tools (e.g., CVS, Subversion) with popular bug tracking system (e.g. Bugzilla). Scmbug offers a policy-based mechanism of capturing and handling integration of SCM events, such as committing software changesets and labeling software releases, with a bug-tracking system [35].

In [35], some advanced features of Scmbug are supported:

- Commit software changesets and label software releases with a bug-tracking system.
- Synchronous verification checks between SCM tools and bug-tracking system.
- Secure deployment over the public Internet.
- An interface to integrate SCM tools with bug-tracking system.
- Version Description Document (VDD) generator used to report the defect, and what changed in components according to that defect between two releases of a software.
4.2.2 System Architecture

Figure 13: Scmbug system architecture

Figure 13 [35] shows the system architecture of Scmbug, which is a client/server system. Firstly, the pre-commit hooks are adopted to capture the standard Subversion events. Later, a generic glue mechanism is made to handle these events, which are translated into integration requests and transported to a server daemon. Finally, the daemon parses above data information that Bugzilla can process.

4.2.3 Integration Daemon

4.2.3.1 Public Internet Deployment

As we mentioned before, secure deployment of the integration over the public Internet is one of the advanced features of Scmbug. Some security techniques used in Scmbug are listed below:

- Minimizing exposure. Bugzilla works only with MySQL as a database backend. It is therefore necessary to access the database with the purpose of integration. However, opening the MySQL TCP port over the public Internet is a serious security risk since it exposes access to other applications running on the database system [35]. Scmbug employs a separate integration daemon (see figure 13) to overcome this problem by only exposing an integration interface and keeping the TCP port closed.

- Integration security mechanism. As seen in figure 13, Scmbug integration daemon employs a public port to receive the translated integration requests without identifying if the requests are from the trusted user. Therefore, a username mapping mechanism between SCM tools and bug-tracking systems is obligatory. I.e., only the username that passes the identification in both systems can be trusted and recognized. Moreover, in order to avoid phony integration requests by accident, public key authentication can be used to identify the integration requests between the glue and the integration daemon.

4.2.3.2 Public Integration Interface

Currently, there is no public interface implemented in the popular bug-tracking systems (e.g., Bugzilla). Integration efforts currently duplicate functionality implemented in the codebase of the bug-tracking system [35]. However, when the bug-tracking system database is changed, the glue code in all the SCM repositories needs to be synchronized for the integration by the manual effort
which could be at risk of introducing the errors. One possible solution to resolve this problem is to directly reuse the functionality in the bug-tracking system database by using a variable (installation_directory) in the integration daemon which points to the location of resource distribution of the bug-tracking system.

4.2.4 Integration Policies

In [35], some integration policies of Scmbug are discussed as below:

- Presence of distinct bug IDs. Scmbug uses the predefined log message template including a bug id to commit a changeset (see figure 14).

  SCMBUG ID:
  SCMBUG NOTE:

  Figure 14: Scmbug log message

- Valid log message size. An optional policy to verify the commit log message meets a configurable minimum log message size limit, which help to facilitate a detailed log message commitment.

- Valid bug owner. A mapping mechanism to ensure only the owner of the bug id specified in the bug-tracking system can commit a changeset.

- Open Bug State. A verification check is applied to ensure that the status of bug id is open before committing any changeset. For instance, in Bugzilla, the statuses for the resolved and finished bug id are noted as FIXED and INVALID respectively. Accordingly, the commitment against the above bugs cannot be handed in and should be noticed as a consequence.

- Valid Product Name. It is a double check mechanism to make sure that the bug id is really associated with a product name specified in the bug-tracking system. The integration action only works when they can perfectly match with each other.

- Convention-based Labeling. A label naming policy is introduced in order to ensure the labeled name match a configurable format which is defined by using regular expression.

4.2.5 Version Description Document Generator

Version description document (VDD) is one of the advanced features of Scmbug, which can report the internal change relations between software releases. Although some SCM tools (e.g., Subversion) can get changelog information between different revisions by using command (e.g., ‘svn log –r <rev_old>:<rev_nev>’), the generated changelog is just an overly detailed accumulation of log message, a low-level source changes. Nevertheless, VDD can facilitate a more intuitive expression by reporting a high-level abstraction of defects and their associated changes between different software releases.

4.2.6 Integration Example

Here, we use the example in [35] to show an actual integration sequence by using CVS and Bugzilla. Compare to CVS, there are two things needed to be addressed when using Subversion.
First, Subversion uses the command ‘svn commit’ instead of ‘cvs commit’ when send changes from working copy to the repository. Second, Subversion is not yet capable of using a predefined log template when the user’s editor is opened to enter a log comment [36]. Therefore, user needs to enter the predefined log template manually.

This example shows a bug, which ID is 417. The root problem of Bug 417 is shown in its comment #6 in Bugzilla, see figure 15. Later, a developer makes a fix and commits the changeset to the repository. As we mentioned before, Scmbug uses the predefined log message template including a bug id to commit a changeset, which indicates the bug ID and detailed component changes, see figure 16. Figure 17 presents the log comment stored in CVS for Bug 417. After integration, this CVS log message and a list of affected files are also documented in Bug 417, comment 10 in Bugzilla, shown in Figure 18. From above approach, we can easily identify why the changeset occurred and what changed of a particular bug.

Figure 15: Root of the problem documented in bug 417, comment 6 in Bugzilla. The problem is not documented in CVS

Figure 16: Documenting a fix for bug 417 during a ‘cvs commit’

Figure 17: Log comment entered in CVS for Bug 417. Through integration, this CVS entry is also documented in Bug 417, comment 10 in Bugzilla, shown in Figure 18

Figure 18: Bug 417, comment 10 in Bugzilla. The reason why this changeset occurred was documented 2 days earlier in comment 6, shown in Figure 15
5 Conclusions

Software configuration management (SCM) is one of the basics of good software development. The categories of SCM include many areas. Change management is one of the important parts within SCM, since change is the essential fact of software system. Change management and error management are two highly related development activities. In order to facilitate above activities more efficiently, many SCM and error management tools are designed and proposed as a consequence. This report aims to give a systematical overview of some representative tools rather than to explain how to use them in practice. Hopefully, this information can help users to have a thoroughly understanding of the basic models, functions, characteristics, features of these tools and how they manage the different activities. It is always nice and important to use tools with a theoretical understanding of their concepts backgrounds behind, since it can help users and organizations to identify the most suitable tools when applying SCM and error management in their software development process. Additional, a desired analysis of the possible integration of SCM tool (e.g., Subversion) and bug-tracking system (e.g., Bugzilla) is elaborated for the sake of a completed change management. Finally, a set of practical exercises based on RCS are developed that help to illustrate some simple functions and their modification to facilitate RCS change management, presented in Part II.
Part II Practical Exercises

In order to have a comprehensive and impressive understanding of the SCM, we develop a set of exercises that help to illustrate simple SCM functions and their modification to support the change management. The exercises are based on RCS open-source. For the detailed design and source code, please refer to the design description document.

1 Exercise 1

RCS deals with revisions in a file-oriented way, whereas Subversion handles versioning of directories as well as files in a typical hierarchy structure. This is a significant improvement beyond RCS, since it allows copying, moving, renaming and other rearrangements of files. The aim of this exercise is to create a command by imitating the Subversion checkout way, which checks out all the versioned files under the source directory from their RCS folders to the same level directory position at once. Figure 19 shows the situation before and after the cotree command issue in the case of RCS folders in the source directory. The implementation of this exercise can be based on RCS co command with the same option.

![Figure 19: A simple graphic illustration of the cotree command in the source directory](image)

Command Description:

Syntax:

Parameters:

[Directory]: specifies the directory of the check out files. If omitted, the current working directory will be used.

-{filMpqru}[rev] -ddate -jjoins -ksubst -sstate -T -w[who] -Vn -xsuff --zone: this syntax is used because the cotree command invokes co command with the same option.

file: specifies the files need to be checked out.

Main Scenarios:

Example 1: \texttt{cotree g:\thesis -l *}

This scenario will check out the latest version of all the RCS files under the directory g:\thesis.

Example 2: \texttt{cotree -sStable 1.txt 2.txt}

This scenario will check out the latest version of RCS file 1.txt and 2.txt that has the status \textit{Stable} under the current working directory.

\section{Exercise 2}

For the software configuration management (SCM) tools, it is typical to copy the project structure from the server to the user’s own working directory. This can support the parallel development and enable multiple developers to work simultaneously. The aim of this exercise is to create a command, which help to copy the source directory structure to the destination directory excluding the RCS folders, but instead, automatically generate RCS files containing the absolute path in the form of a single line of text which refer to the source directory RCS folders. Figure 20 is a graphic illustration of how it works.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{exercise2.png}
\caption{A simple graphic illustration of the exercise 2}
\end{figure}
Command Description:

Syntax:

\texttt{costruct sourceDirectory [destinationDirectory]/[.]/[\]}\

Parameters:

\texttt{sourceDirectory}: specifies the source directory of the structure copy.

\texttt{[destinationDirectory]/[.]/[\]}: specifies the destination directory which put the created copied structure in. If omitted or denoted as ‘.’, the current working directory will be used as the destination directory by default.

Main Scenarios:

Example 1: \texttt{costruct g:\thesis s:\SCM}

This scenario will copy the structure under the path g:\thesis to the s:\SCM and automatically generate RCS files containing the absolute paths which refer to the RCS folders under g:\thesis.

Example 2: \texttt{costruct g:\thesis .}

This scenario will copy the structure under the path g:\thesis to the current working directory and automatically generate RCS files containing the absolute paths which refer to the RCS folders under g:\thesis.

Example 3: \texttt{costruct g:\thesis}

This scenario has the same result as previous “\texttt{costruct g:\thesis .}” does. It will copy the structure under the path g:\thesis to the current working directory and automatically generate RCS files containing the absolute paths which refer to the RCS folders under g:\thesis.

3 Exercise 3

In exercise 1, you have created a \texttt{cotree} command by imitating the Subversion \texttt{checkout} way, which checks out all the versioned files under the source directory from their RCS folders to the same level directory position at once. For this exercise, you need to modify your existing \texttt{cotree} command to fit a new requirement that checks out all the versioned files under the destination directory from their RCS files to the same level directory position at once. Figure 21 shows the situation in the case of RCS files in the destination directory. The implementation of this exercise can be based on \texttt{cotree} command with the same option.
Figure 21: A simple graphic illustration of the `cotree` command in the destination directory

Command Description:

Syntax:

```

```

Parameters:

- **[Directory]**: specifies the directory of the check out files. If omitted, the current working directory will be used.
- `-{flMpqru}[rev] -ddate -jjoins -ksubst -sstate -T -w[who] -Vn -xsuff -zzone`: this syntax is used because the `cotree` command invokes `co` command with the same option.
- **file**: specifies the files need to be checked out.

Main Scenarios:

Example 1: `cotree s:\SCM -l *`

This scenario will check out the latest version of all the RCS files under the directory `s:\SCM`.

Example 2: `cotree -sStable 1.txt 2.txt`
This scenario will check out the latest version of RCS file 1.txt and 2.txt that has the status Stable under the current working directory.

4 Exercise 4

Change is the essential fact in a large, complex software system. Therefore, change management is particular important and necessary to ensure the software changes in a controlled way. Within a company, especially the well-organized big company, they set their own change management procedures and process based on the SCM standards and also incorporated into their quality handbook. The change management process includes several stages. The first is to fill the change request (CR). Normally, the CR is used to record the details of the change request, like the reason why need to change and how to change. However, the real CR format can be different according to the definition of different companies or the SCM planning for specific projects. Figure 22 shows a sample change request format. A change request is a text file, which is defined as three parts. The first part indicates Title, Change requester, Initiate Date and Priority. The second part shows the files which need to be changed. The third part includes a description of the change. The aim of this exercise is to create a command crcre, which initiates the CR by following the sample format and checks in the CR in the CR library as a versioned file under RCS control. (Hints: the syntax of command crcre could be crcre [-Ddir] [-ttitle] [-ffile] [-ppriority] CRname... Here, the -D option is used to indicate the CR library directory. If omit -D option, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently. If any of the -t -f or -p option is not indicated, a guide question will lead the user to fulfill the CR alternatively as well as the description part. Moreover, the Change Requester and the Initiate Date in the CR should be generated automatically and in accordance with the RCS keywords.)

Figure 22: A sample change request (CR)

Command Description:

Syntax:

```
crcre [-Ddir] [-ttitle] [-ffile] [-ppriority] CRname...
```
Parameters:

[-Ddir]: specifies the CR library directory where the CR will be checked in later as a versioned file under RCS control. If omitted, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently.

[-ttile]: specifies the title of the change request. If omitted, a guide question will lead the user to fulfill the title later. Note: The title must be quoted if it contains spaces.

[-ffile]: specifies the file need to be changed of the change request. If omitted, a guide question will lead the user to fulfill this part later.

[-ppriority]: specifies the priority of the change request. If omitted, a guide question will lead the user to fulfill the priority later.

CRname: specifies the change request name.

Main Scenarios:

Example 1: crcre -Dg:\thesis –t"Implement change management” -ftest.txt -ffunction.c -plow CR1.txt

This scenario will create a CR as shown in figure 23. Figure 24 shows how the procedure works in the CMD window.

| Title: Implement change management |
| Change Requester: ShLiu |
| Initiate Date: 2008-10-26 13:44:33 |
| Priority: low |
| File need to be changed: |
| test.txt |
| function.c |
| Description: |
| This is the example1 of the main scenario. |
| More scenarios are coming soon. |

Figure 23: The created change request based on scenario 1
Example 2: `crcre CR2.txt`

This scenario will create a CR as shown in figure 25. Figure 26 shows how the procedure works in the CMD window. Since `-D` option is omitted, the program will automatically check the CR environment variable instead. Neither the `-D` option nor the CR environment variable is set, an error message will be shown consequently. If CR environment variable exists, some guide question (see figure 26) will lead the user to fulfill the change request successfully as well as the description part, since the `-t` `-f` and `-p` option are not indicated in the command line.

```plaintext
G:\thesis>crc
c -D g:\thesis -t "Implement change management" -ft est.txt -ffunction.c -plow CR1.txt
enter Description: ‘terminated with single ’.’
This is the example 1 of the main scenario.
More scenarios are coming soon.
.
g:\thesis\RCS\CR1.txt <- g:\thesis\CR1.txt
enter description, terminated with single ‘.’ or end of file:
NOTE: This is NOT the log message!
>> change request CR1
>>
initial revision: 1.1
done
G:\thesis>
```

Figure 25: The created change request based on scenario 2
Exercise 5

In exercise 4, you have created a command `crecre` to initiate the change request (CR) in the CR library. After completing the CR, the next stage of the change management process is to analyze the CR’s validation by the relative responsible committee within some predefined period time. The committee is composed mainly of relative engineers, experts and other responsible people, who have the authority to validate the change request. The change request can only be valid when all the members in the committee agree on that change. Otherwise, it is rejected. Of course, the committee should give the reason why it has been rejected at last. The aim of this exercise is to create a command `crlist`, which easily list the existing CRs’ content for the committee validation. (Hints: the syntax of command `crlist` could be `crlist [-Ddir] [-f] [CRname]...` Here, the -D option is used to indicate the CR library directory. If omit -D option, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently. The -f option is used to list the whole content and its latest status of CRs (see figure 27). As shown in figure 27, there are two parts of the list CRs. The upper part indicates the CR name, the latest version No., the latest status, the latest modified people and the latest modified date of the CR. The lower part shows the latest whole CR content. If omit -f options, it will list the Change Requester, Initiate Date, Priority and Title of the specific CRName (see figure 28). If omit CRName as well, it will list the Change Requester, Initiate Date, Priority and Title of all CRs in CR library (see figure 29).)
Command Description:

Syntax:

crlist [-Ddir] [-f] [CRname]…
Parameters:

[-Ddir]: specifies the CR library directory where the CR will be checked in later as a versioned file under RCS control. If omitted, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently.

[-f]: list the whole content and its latest status of CR

[CRname]: specifies the change request name need to be listed. If omitted, all the CRs in CR library will be listed.

Main Scenarios:

Example 1: crlist -Dg:\thesis -f CR3.txt

This scenario will show the latest CR3 status and list the whole CR3 content. (see figure 27).

Example 2: crlist -Dg:\thesis CR3.txt CR4.txt

This scenario will list the Change Requester, Initiate Date, Priority and Title of the specified CR3 and CR4 (see figure 28).

Example 3: crlist -Dg:\thesis

This scenario will list the Change Requester, Initiate Date, Priority and Title of all the CRs in CR library. If the “CR” environment variable is set, the argument "-Dg:\thesis" can be omitted as well (see figure 29).

6 Exercise 6

As mentioned in exercise 5, a committee will take the responsibility to analyze the CR’s validation after completing the CR. If all the members in the committee agree on that change, and then the CR can be approved. Otherwise, it is rejected. If the CR is approved, we need to change its state from Exp (default state, which means the CR is being processed) to Init (means the CR is just created and no action is made yet). A CR state describes the status of the CR. Any identifier is acceptable for state. Some recommended set of states is Init (means the CR is just created and no action is made yet), Exp (means the CR is being processed), Term (means the CR is completed and no more updating is expected) and Rel (means the CR is integrated in a release). Others like Impl (means CR is done, but not yet tested), Test (means the modifications described in the CR are tested) and Post (means the CR is postponed for a later release). Figure 30 shows a general CR states process. When a new CR is created in gets the Exp state by default. After being approved by the committee, set the state from Exp to Init. Later, each time a CR is updated (a new file is added, or a description is modified), the CR by default gets the Exp state. When a programmer, who is responsible for the implementation of a change, finds that all the changes are done, he/she sets the Impl state. A programmer who has responsibility to test the changes registered in the CR, puts the Test state when the test is done. If a problem was found, the CR returns back to the state Exp. The project leader (or someone else) approves the changes by terminating the CR. The CR gets the Term state, and cannot be changed until the Term state is cancelled. Finally, when changes specified in the CR are integrated in a new product release, the product admin sets the state Rel. The aim of this exercise is to create a command crstate, which easily change the status of the CR. (Hints: the implementation of this exercise can be based on RCS rcs command with the same option –s’state’:[‘rev’].)

38
Command Description:

Syntax:

```
crstate [-Ddir] -s'state':[‘rev’] CRname...
```

Parameters:

- `[-Ddir]`: specifies the CR library directory where the CR will be checked in later as a versioned file under RCS control. If omitted, the program will automatically check the CR environment variable instead. Neither the `-D` option nor the CR environment variable is set, an error message will be shown consequently.

- `–s’state’:[‘rev’]`: set the state attribute of the revision ‘rev’ to ‘state’. If ‘rev’ is a branch number, assume the latest revision on that branch. If ‘rev’ is omitted, assume the latest revision on the default branch.

- `CRname`: specifies the change request names need to be changed states.

Main Scenarios:

Example 1: `crstate -Dg:\thesis -sInit:1.1 CR4.txt`

This scenario will change the status of CR4 revision 1.1 to Init state.

Example 2: `crstate -Dg:\thesis -sImpl CR4.txt`

This scenario will change the status of the latest revision on the default branch of CR4 to Impl state.

7 Exercise 7

The change request (CR) can not just be a plain documentation to record the description of the change and why need to change. Moreover, it can reveal the components (files) derivation history associating with that particular CR. The aim of this exercise is to improve the RCS `ci` command performance by adding `-c` option. That is: when check in a component (file), refer to its associated CR and automatically update it. Figure 31 and figure 32 shows a comparison before and after issuing this performance improvement. Comparing to figure 31, we can see that figure 32 clearly shows the track of the components (files) derivation history, which may be efficiently used for related project manager, engineers and programmers to supervise and look through the CR process and status. (Hint: When the user checks in files, he/she can also specify a CR that relates to the check-in file. The `ci` command will first check in the corresponding check-in file, and then check out CR, automatically update (the modified version No. history of the component (file), see figure 32 “File need to be changed” part) and append the component (file) information (e.g., latest version No., latest modification person, latest modification date and log message) in that CR, and
check it back to the CR library later. In more details, if the CR does not include the file name (see figure 31 “File need to be changed” part) of the check-in file, a suggestive question (e.g., “The file does not exist in the specified CR, do you really want to add it?”) will show up to help the user to make decision. If the user choose yes, then the file will be appended at the end of the part “File need to be changed” in CR.)

Figure 31: A sample CR before applying `ci -c` option

| Title: improve the RCS "ci" command performance by adding ":c" option |
| Change Requester: ShLiu |
| Initiate Date: 2008-11-16 09:43:34 |
| Priority: high |
| File need to be changed: |
| - ci.c |
| - conf.h |
| - rcsbase.h |
| - document.txt |
| Description: |
| - Improve the RCS "ci" command performance by adding "-c" option. |
| - That is: when check in a component (file), refer to its associated CR and automatically update it. |

--------
Figure 32: A sample CR after applying `ci` `-c` option

Command Description:

Syntax:

```
ci -{-fI|Mpqru}\[rev\] -ddate -jjoins -ksubst -sstate -T -w[who] -Vn -xsuff -zzone [-Ddir] [-c'CRname'] file...
```

Parameters:

```
```

```
[-Ddir]: specifies the CR library directory where the CR will be checked in later as a versioned file under RCS control. If omitted, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently.
```
When the user checks in files, he/she can also specify a CR that associates the check-in files. It will automatically update (the modified version No. history of the check-in file, see figure 32 “File need to be changed” part) and append check-in file information (e.g., latest version No., latest modification person, latest modification date and log message) in that CR.

**file** specifies the files need to be checked in.

**Main Scenarios:**

**Example 1:** `ci -Dg:\thesis -cCR8.txt document.txt`

Figure 33 shows how the program works in the CMD window based on the example scenario 1. When check in the file `document.txt`, it refers `CR8.txt` by `ci -c` option as well. After adding the log message of `document.txt`, the program will check out `CR8.txt`, update information in it and check it back to the CR library. The result `CR8.txt` is shown in figure 34. We can see that there are two parts changed comparing the original `CR8.txt` in figure 31. One is in the “File need to be changed” part, it adds the latest version No. of `document.txt`. Another part is that it appends the `document.txt` information (e.g., latest version No., latest modification person, latest modification date and log message) at the end of `CR8.txt`.

![CMD window](image)

Figure 33: A CMD window shows how the example 1 scenario works
Figure 34: The result CR based on example 1 scenario

Example 2: `ci -Dg:\thesis –c CR8.txt ci.c document.txt`

Figure 35 shows how the program works in the CMD window based on the example scenario 2. This time it check-in two files when referring `CR8.txt` by `ci -c` option. First, it will check-in `ci.c` and update the information in `CR8.txt`. Later, it will check-in `document.txt` and update the information in `CR8.txt` as well. The result `CR8.txt` is shown in figure 36.
Figure 35: A CMD window shows how the example 2 scenario works
<p>| Title: improve the RCS &quot;ci&quot; command performance by adding &quot;-c&quot; option |
| Change Requester: ShLiu |
| Initiate Date: 2008-11-16 09:43:34 |</p>
<table>
<thead>
<tr>
<th>Priority: high</th>
</tr>
</thead>
<tbody>
<tr>
<td>File need to be changed:</td>
</tr>
<tr>
<td>ci.c 3.11</td>
</tr>
<tr>
<td>conf.h</td>
</tr>
<tr>
<td>rcsbase.h</td>
</tr>
<tr>
<td>document.txt 1.3 1.15</td>
</tr>
<tr>
<td>Description:</td>
</tr>
<tr>
<td>improve the RCS &quot;ci&quot; command performance by adding &quot;-c&quot; option.</td>
</tr>
<tr>
<td>That is: when check in a component (file), refer to its associated CR and automatically update it.</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>document.txt: 1.3 (ShLiu 2008-11-19 12:15:11)</td>
</tr>
<tr>
<td>add the new requirement that if the CR does not include the file name of the check-in file, a suggestive question (e.g. &quot;The file does not exist in the specified CR, do you really want to add it?&quot;) will show up to help user to make decision.</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>ci.c: 3.11 (ShLiu 2008-11-19 16:55:17)</td>
</tr>
<tr>
<td>add changemanagement function in ci command</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>add the description of function changemanagement</td>
</tr>
<tr>
<td>-----</td>
</tr>
</tbody>
</table>

Figure 36: The result CR based on example 2 scenario

Example 3: `ci -Dg:\thesis -cCR8.txt testcase.c document.txt`

Figure 37 shows how the program works in the CMD window based on the example scenario 3. This time it check-in a new file `testcase.c` together with `document.txt` when referring `CR8.txt` by `ci -c` option. However, from the original `CR8.txt` shown in figure 31, `testcase.c` doesn't exist in the `CR8.txt`. Therefore, the system asks the user to decide whether to add it on the “File need to be changed” list of `CR8.txt` or not. If choose yes, the related `testcase.c` information will automatically update in `CR8.txt`. Since we have more than one check-in file, it will continuously update the `CR8.txt` until all the check-in files finished. The result `CR8.txt` is shown in figure 38.
Figure 37: A CMD window shows how the example 3 scenario works
8 Exercise 8

In exercise 7, you have improved the RCS `ci` command by adding `-c` option which can sufficiently track the components (files) derivation history with their related CRs. By doing that, there is a possibility to check which CR is included in a particular product release. Because the new release product is generated directly from the versioned components (files), which derivation history has already been registered in CRs. The aim of this exercise is to create a command `crlink` showing the relation between the components (files) and their related CRs. Figure 39 shows a list of changed file and their related CRs. The **File** column shows the changed file name and path. The **Ver**, **State** and **User** column shows the latest version No. status and modification user of the changed file respectively. The **relation** column shows the relation between the changed file and CRs. If the changed file latest version No. corresponds to the highest Version No. in its associated CRs, then it is denoted as “=”. However, if the changed file latest version No. bigger or smaller
than the highest Version No. in its associated CRs, then it is denoted as “>” or “<”. Moreover, if the changed file does not been included or registered in any CRs, it is denoted as “!”.

The last **Change Requests** column shows detailed information about all the changed file versions with respect to different CRs.

---

<table>
<thead>
<tr>
<th>File</th>
<th>Ver</th>
<th>Status</th>
<th>User</th>
<th>Relation</th>
<th>Change Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>.crlink\code\header\RCS\conf.h</td>
<td>1.2</td>
<td>Exp</td>
<td>ShLiu</td>
<td>&gt;</td>
<td>1.1:Crl1.txt</td>
</tr>
<tr>
<td>.crlink\code\source\RCS\cli</td>
<td>3.12</td>
<td>Exp</td>
<td>ShLiu</td>
<td>&gt;</td>
<td>2.1:Crl1.txt, 3.11</td>
</tr>
<tr>
<td>.crlink\doc\RCS\document.txt</td>
<td>1.4</td>
<td>Exp</td>
<td>ShLiu</td>
<td>=</td>
<td>1.3:Crl1.txt</td>
</tr>
<tr>
<td>.crlink\doc\req\RCS\requirement.txt</td>
<td>1.0</td>
<td>Exp</td>
<td>ShLiu</td>
<td>&gt;</td>
<td>1.5:Crl3.txt</td>
</tr>
<tr>
<td>.crlink\RCS\readme.txt</td>
<td>2.0</td>
<td>Exp</td>
<td>ShLiu</td>
<td>&gt;</td>
<td>1.7:Crl1.txt</td>
</tr>
<tr>
<td>.crlink\test\RCS\test.txt</td>
<td>1.2</td>
<td>Exp</td>
<td>ShLiu</td>
<td>-</td>
<td>1.2:Crl2.txt</td>
</tr>
<tr>
<td>.crlink\test\RCS\testcase.e</td>
<td>1.6</td>
<td>Exp</td>
<td>ShLiu</td>
<td>-</td>
<td>1.5:Crl3.txt</td>
</tr>
<tr>
<td>.crlink\test\RCS\testcase.e</td>
<td>1.6</td>
<td>Exp</td>
<td>ShLiу</td>
<td>-</td>
<td>1.5:Crl3.txt</td>
</tr>
</tbody>
</table>

Figure 39: A list of changed file and their related CRs

**Command Description:**

**Syntax:**

crlink [-Ddir] [-ddir] …

**Parameters:**

- **[-Ddir]:** specifies the CR library directory where the CR will be checked in later as a versioned file under RCS control. If omitted, the program will automatically check the CR environment variable instead. Neither the -D option nor the CR environment variable is set, an error message will be shown consequently.

- **[-ddir]:** specifies the file directory. If omitted, the current working directory will be used by default.

**Main Scenarios:**

**Example 1:** crlink -Dg:\thesis –dg:\crlink

This scenario will show the relations between the files under the path g:\crlink with their related CRs under the CR library directory g:\thesis. As shown in figure 39, there are eight files in different subsystem path. The last two columns show the relation between the latest file version and its registered version within associated CR.
References


