Durham E-Theses

A process model of maintenance with reuse: an investigation and an implementation abstract

Kwon, Oh Cheon

How to cite:

Use policy

A Process Model of Maintenance with Reuse:

An Investigation and an Implementation

Oh Cheon Kwon

Ph.D. Thesis

University of Durham

Department of Computer Science

22 December 1997

Oh Cheon Kwon
A Process Model of Maintenance with Reuse:
An Investigation and an Implementation
Abstract

Sixty to eighty per cent of the software life-cycle cost is spent on the software maintenance phase because *software maintenance* is usually more difficult than original development and legacy systems are generally large and complex. *Software reuse* has recently been considered as a best solution to enhance the productivity of a software development team and to reduce maintenance costs. In addition, *Software Configuration Management (SCM)* is a central part of software maintenance as it is associated with changing existing software and is a discipline for controlling these changes. Thus, both software reuse and SCM have been proposed for making a significant improvement in productivity, quality and cost. However, so far these two technologies have been investigated separately. In order for software reuse and SCM to produce effects by synergy, both approaches require to be introduced into a maintenance environment together. Since software reuse and SCM, and software reuse and software maintenance have many similarities in their activities, these disciplines can be integrated within a software maintenance environment. This research has therefore developed an integrated process model for 'Maintenance with Reuse (MwR)', that supports SCM for a reuse library which is actively maintained for use in a software maintenance environment.

This thesis addresses an integrated process model called the MwR model and its prototype tool TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library) that consist of a configuration management (CM) process, reuse process, maintenance process and administration of a reuse library. The MwR model and TERRA provide reusers and maintainers with many activities of these four processes such as classifying, storing, retrieving, evaluating, and propagating reusable components, including controlling changes to both reusable components and existing systems.

The process model of an integrated approach has been developed and validated using Process Weaver. The TERRA tool has been implemented on the WWW so that the prototype can provide portability, traceability, integration with existing tools, and a distributed maintenance environment. The TERRA prototype has been tested and evaluated through a scenario based case study. Several scenarios based on real data have been created and used for the case study so that an organisation can apply the model and tool to its maintenance environment without many problems.

The software maintenance community is facing serious problems with legacy systems, such as a ever increasing frequency of changes and backlogs, lack of integrated tools and methods, and lack of software maintenance support environments. The control and management of changes to the software components in a reuse repository are crucial to successful software development and maintenance. If the component is being used in multiple systems effects of uncontrolled change are more critical. However, reuse libraries and servers currently available have not been successful as they do not support further development or maintenance of the reusable components. In addition, most of them are not sophisticated since they have not been linked to a development/maintenance environment.

The integrated model of MwR can overcome many problems that exist in software maintenance and reuse through introduction of SCM functionalities into a maintenance environment. Thus, the integration of these common activities will greatly contribute to enhancing the productivity and quality of software, and will additionally lead to reducing the costs and backlogs of changes within a maintenance environment.

Acknowledgements

A number of people have helped me to finish this work. Without their support, I could not have completed this research. I would firstly like to thank my supervisors, Dr. Cornelia Boldyreff and Malcolm Munro, for their guidance and encouragement over the last three years. Their comments and advice have been invaluable during the writing of this thesis.

I am grateful to two examiners, Professor Ian Sommerville and Professor Keith Bennett for sparing their time to review my dissertation and to make valuable comments and suggestions.

I would also like to thank other members of the Department of Computer Science, especially, Steven Glover, David Nelson, Deborah Robson and Louise Hudson for their support.

Many thanks and love go to my family, especially, my wife Eun-Hee, my daughters Yu-Jin and Ye-Na, my parents-in-law, Rev. and Mrs. Uhm, for their huge support, patience and encouragement throughout this research.

I acknowledge the financial support from the British Council, SERI (Systems Engineering Research Institute) and my brothers in Korea.

Declaration

The material contained in this thesis has not been previously submitted for a degree in the University of Durham or any other university.

Statement of Copyright

The copyright of this thesis rests with the author. No quotation from it should be published without his prior written consent and information derived from it should be acknowledged.

Contents

List of Figures

xiii

Chapter 1

Introduction

1.1 Overview of Software Engineering

In the 1960s, as computer hardware was developed greatly, computer systems became capable of processing complex and large computer applications, leading end users to produce high volumes of end users' requests for software systems. In order to solve these problems, the concept of an engineering discipline has been introduced to the field of software development.

The origin of *Software Engineering* goes back to the late 1960s when a conference was held to discuss what could be termed the 'software crisis'. This software crisis originated from the emergence of third-generation computer hardware that was more powerful than second-generation machines and enabled software developers to build large software systems. The existing methods and techniques of software development were not good enough to produce a large software system. Software production was facing a crisis as many large projects were overdue and over budget, and on top of that, software systems developed were unreliable and difficult to maintain. Thus, to solve these problems with developing large software systems, the software industry needed new methods and techniques. Software Engineering

is different from other engineering disciplines in that software products produced from Software Engineering do not have any physical form. Software Engineering like other engineering is not just concerned with producing software products but producing products in a cost effective way [100].

The first International Conference on Software Engineering was held in 1973, and the *IEEE Transactions on Software Engineering* were firstly published in 1975. Since these two events, many academic institutions now offer a Master of Software Engineering degree. Bauer [5] who is one of the early leaders in the field of this new subject, defines Software Engineering as follows:

> *The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines.*

Software Engineering disciplines concerned with this research are *software maintenance, software reuse, Software Configuration Management.* These three fields have long been recognised as crucial factors to be considered in order to improve software quality and to enhance software productivity. These disciplines are all very closely connected with the software development/maintenance environment. In other words, in order to make these fields effective in a real environment, methods and tools related to them must be linked to the development/maintenance environment and also implemented in it.

Software maintenance has been defined as:

> *The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [60].*

Informally, software maintenance contains all activities that take place after a software product has been delivered to the end user, but a more formal definition

has recently been made by Bull and Bennett [20] as follows:

Software maintenance is the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way.

The cost of maintaining a legacy system amounts up to 60-80 percent of all effort that a software development organisation spends during the system life-cycle. As an organisation can own more software systems over time, these systems are gradually getting older, and every system is subject to change so much during the software life-cycle, the cost of maintenance will be increasing greatly. The changes to an existing system are inevitable because of the following causes:

- After delivery, a software product is subject to modification in order to improve performance or provide new functionalities.

- After delivery, a software product is subject to modification for adapting a system to change in the execution environment.

- After delivery, a software product is subject to modification for fixing abnormal results or errors.

- After delivery, a software product is subject to modification for preventing problems, for instance, improvement of software maintainability.

Most efforts in the software industry have been on software development whose objective is to produce a product that is on time and within budget while meeting user requirements, not a product that is reliable and maintainable. For these reasons, maintenance has to be dealt with, but maintenance is usually more difficult than original development. To tackle these problems with software maintenance, we therefore need to investigate some Software Engineering disciplines such as *software reuse* and *Software Configuration Management (SCM)*.

3

Software reuse has been defined as [9]:

> *The reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of the other system.*

Biggerstaff and Perlis [9] have made it clear in their discussion that they do not expect such reuse to occur incidentally– it must be planned for and capitalised upon. To make reuse effective, reuse should be supported by the *'Development for Reuse(DfR)'* process that aims at creating reusable software or re-engineering existing software to extract reusable software from an existing system and the *'Development with Reuse(DwR)'* process whose aim is to develop a new system using reusable components.

Software Configuration Management (SCM) has been defined as:

> *The discipline of 'identifying' the configuration of a system at discrete points in time for purposes of systematically 'controlling' changes to this configuration and 'maintaining' the integrity and traceability of this configuration throughout the system life cycle* [8].

SCM is responsible for coordinating and controlling software development, especially, maintenance activities through the SCM principles of *configuration identification, configuration control, configuration status accounting, configuration audit* in order to attain and maintain product integrity.

1.2   Objectives of Research

The research in this thesis is associated with an integrated approach of three research fields: *software reuse, Software Configuration Management (SCM) and software maintenance.* This approach is relatively new because although these research fields

4

have been recognised as important over the last decade, they have generally been investigated separately.

Of all the known software technologies, software reuse has the best potential for reducing software costs and enhancing the productivity of software development; nevertheless, serious barriers must be eliminated before reuse can take its place as a major software technology. Jones [62] raised a number of engineering problems associated with reusing software as follows: creating or recognising a potentially reusable component; cataloguing and retrieving them from a repository efficiently; and finally, how to compose complex systems from those components and how to control the change of reusable components. The last two problems above can be solved through an SCM process.

Although some reuse libraries have been developed and announced, there is no reuse library that can support a software maintenance environment and provide the functionalities of change control and version control for reusable components within a reuse library. This research seeks to develop the Maintenance with Reuse (MwR) model that supports a maintenance environment through the process of reuse, and also controls the evolution of a reusable software library. The aim of this research is to present solutions to tackle many problems with software maintenance and reuse through the integrated model of software maintenance and reuse within a Software Configuration Management environment.

1.3   Research Method and Criteria for Success

This research will implement a rapidly built prototype and enhance it by solving potential problems such as functionality, performance, ease of use, applicability and maintainability [15]. Therefore, this research will use an evolutionary approach as a development method. Two versions of the prototype will be implemented through evolutionary development. The MwR model and prototype produced in this research

are employed and evaluated through a scenario based case study. The existing reuse libraries will be evaluated and the results from evaluation used for the development of an initial prototype.

The major criteria for success depends on attaining the following expected results:

- To develop models for 'Maintenance with Reuse (MwR)' that supports the evolution of a legacy system by a reuse library. Change requests (CRs) issued from maintainers or end users can be implemented by a reuse process.

- To develop the procedure of change and version control for reusable components within a reuse library. Within a maintenance environment, the process by which a change to reusable components is issued, evaluated, approved or rejected, scheduled and tracked, should be addressed. Modified reusable components could be entered into the library as new versions of these components, since they might be reused. The control and notification procedures for the revisions of a reusable component should be established. Thus, in order for the reuse library to support a legacy system, the library can be well populated and controlled by SCM functionalities.

- To model and implement administrative functionalities associated with a reusable software library. The functionalities of the reuse library include classification, registration and retrieval of reusable components, and notification of changes of reusable components.

- To produce information (i.e., status accounting) related to reuse and SCM. Each reusable component should contain some information including a clear specification, quality and administration (i.e., a change history report, a reuse history report, reuser's evaluation report, etc.).

- To develop a prototype that supports the 'Maintenance with Reuse (MwR)' model. The prototype named TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library) provides users with an automated tool that

6

supports a maintenance process, a reuse process, and an SCM process.

1.4 Outline of Thesis

This thesis is organised as follows:

Chapter 2 presents a survey of software reuse, Software Configuration Management (SCM) and software maintenance since this research is associated with those three fields of software engineering.

Chapter 3 discusses the rationale of this research and includes the history of evolution of the reuse concept, problems with reuse and maintenance, similarities between reuse and SCM and between reuse and maintenance, research directions to be tackled and the originality of this work.

Chapter 4 gives a description of the model that has been developed for this research, and supports three processes, i.e., the Configuration Management (CM) process, the reuse process, and the maintenance process. In addition, the chapter presents the relationships between the reuse process and the maintenance process, followed by the concept of a product line and the administrative function of the reuse library.

Chapter 5 describes TERRA's interaction with an SCM tool, CGI (Common Gateway Interface) and Web server, and tools adopted for implementation of the MwR model.

Chapter 6 depicts the procedures for operation of TERRA, showing the fill-out forms for registration and retrieval of reusable components, and the entry forms for change requests and change approval, etc.

In Chapter 7, the MwR model and its prototype called TERRA are brought together, by applying four scenarios to the model and tool. The data used in this

scenario based case study has been used for construction of a Graphic User Interface (GUI) framework by the object-oriented community.

Chapter 8 evaluates the MwR model and tool using results obtained from the case study in the previous chapter, describing modifications and customisation of the MwR model and TERRA.

Finally, Chapter 9 concludes the thesis by summarising the results from this research work and drawing conclusions of the thesis, followed by further work to be done.

Chapter 2

Background of this Research

This chapter examines the state of the art for three research fields, i.e., software reuse, Software Configuration Management (SCM) and software maintenance. This literature survey focusses on describing basic and extended concepts of each field, their evolution, relevant activities and models, and discussing their techniques and methods, including the identification of similarities and relationships between the three fields.

Section 2.1 describes the evolution of reuse, a process of reuse, review of reusable software libraries currently available, and a software interconnection model and language that can be used for the composition of reusable components, followed by an investigation of the relationship between software reuse and maintenance. Section 2.2 concentrates on developments, activities, models and automation of SCM, identifying SCM related problems within a software maintenance environment. Section 2.3 reviews maintenance models and a software maintenance environment. Section 2.4 describes relationships between reuse, SCM and software maintenance. Finally, Section 2.5 concludes the chapter by summarising the background research focussed on software reuse, SCM, and maintenance, emphasising the need for an integrated approach of these three fields.

2.1 Software Reuse

2.1.1 Introduction

The software community has placed a great deal of emphasis on productivity, shortening lead time, and on product and process quality. In general, *software reuse* has been defined as the use of a given piece of software in the solution of more than one problem. Software reuse has been developed as a new paradigm in the field of software engineering since McIlroy proposed the idea of a software component catalogue from which software assets could be assembled in 1967 [82]. However, *ad-hoc software reuse* has usually been practised by individuals and small groups in many organisations. In recent years, it has been realised that the genuine benefits of reuse can only be achieved through *systematic reuse*, which is domain focused and supported by process assessment and improvement. Since 1991, software reuse has been investigated in large projects, such as REBOOT (REuse Based on Object-Oriented Techniques), SER (Software Evolution and Reuse) and RECYCLE (Application understanding and structural analysis tools) [16].

Software reuse will be one of the major sources of saving software development cost and increasing software quality over the next 15-20 years. By reusing reusable components that have already been developed, an organisation enhances its possibilities to improve both the productivity and the quality of the produced software. We can expect the main benefits from reuse in terms of software quality, productivity, economic return, and lead time (i.e., time-to-market). Several successful reuse experiences have been reported in [71], [75], [81], [90].

The more times a component is reused, the more defects are detected, resulting in higher quality. In addition, if the reusable components are already documented and tested, the new product to be produced using these components requires less work, leading to the enhancement of productivity. Reuse can also improve a product's maintainability and reliability, thereby reducing maintenance labour costs. To

estimate the costs and benefits of reuse, Hewlett-Packard (HP) has developed an economic analysis method which has been applied to multiple reuse programmes in the organisation [75]. Lead time is possibly the most important factor in today's rapidly changing technological environment. The marketing lifetime of a product is steadily decreasing, and requests for rapid changes and extensions to products are increasing. Developing new products by combining available components, i.e. 'Development with Reuse (DwR)', is a way to reduce time-to-market [63].

This section is organised as follows. In Section 2.1.2, the development of reuse is summarised, ranging from the initial use of algorithms, reusable code and packages through REusable Object (REO) to Reusable Software Engineering (RSE). Section 2.1.3 discusses some of the current reuse models and their activities, 'Development with Reuse' based on 'black box' reuse and 'white box' reuse, and the Reuse Capability Model (RCM). A reusable software library, and a software interconnection model and system modelling language are described in Sections 2.1.4 and 2.1.5, respectively. A reuse library is required to classify, store, and retrieve reusable components. A system modelling language is used for composing a system from reusable components. Finally, Section 2.1.6 presents the relationship between software reuse and maintenance.

2.1.2 Evolution of Software Reuse

Reuse has existed from the beginning of software development through the publication of algorithms, the use of high-level programming languages, and the use of packages. Reuse through the publication of algorithms and designs led to the development of computing as an academic discipline. In high-level programming languages, a set of frequently used instructions at the assembly level has been packaged into single constructs at the higher level, which can be used as the form of a notation in order to build a software system. The usefulness of packages depends on some flexibility that can be provided by simple parameterisation or the modification

of sources [102, 49].

In the late 70s, McIlroy's idea was applied in a limited domain by Lanergan and Poynton with excellent results [72]. They identified and classified a lot of code and standard structures that could be used in many of their applications. The initial type of reuse described above was focussed on the reuse of code. However, if we only stick to the reuse of code, we can not gain all the benefits from reuse. Reusable components can include requirement specifications, designs, test plans, test cases, quality assurance checklists, or even the skills and experience of people, as well as program code.

Freeman [42] presented a view called *Reusable Software Engineering (RSE)* which underlies and motivates our interest in the reuse of all information generated in the course of development. The reuse of program code alone has almost no value. It is entirely inconsistent to urge developers to put more effort into the analysis and design activities and not attempt to reuse the information generated there. He described a 5 level hierarchy of types of information that software developers typically need. The levels of reusable information provide a useful set of abstractions that aid our understanding. The types of information are explained as follows. Code fragments like executable code are often viewed as the primary product of the programmer and their effective reuse is one of the oldest objectives of software technology. The level of logical structures includes modules, data collections and the relationships between them (calling, parameter passing, inclusion). The level of functional architectures is related to the external design of a system. This level of information is normally a specification of functions (eg., mathematical subroutine packages and specific packages such as SPSS) and data objects. The level of external knowledge consists of application-area knowledge (scientific laws, mathematical systems, and rules of accounting) and development knowledge (life-cycle models, work-product definitions, test plans, and quality assurance checklists). Finally, environment-level information, the highest level of information, consists of utilisation knowledge and technology-transfer knowledge [42].

12

In the Practitioner project, Boldyreff et al. [13] stated their own particular approach to software reuse: the reuse of concepts rather than code, working with existing software rather than prescribing practices which will lead to the development of new software which is reusable. Their work focused on extracting design concepts from existing code and was successful within a limited domain.

To encompass all of software engineering products, ideas, methods, and principles and to avoid the term software component which is more specific, we call any tangible form of organisational knowledge which can be reused a *REO (REusable Object)* [117].

2.1.3 Reuse Process

In order to make reuse viable and effective, an organisation needs to set up a process of software reuse. Although there are many reuse models, only some of the models are described and discussed below.

Hall [50] proposes that the process of reuse consists of *component engineering (i.e., reverse engineering and Design for Reuse), Design with Reuse, a component library, and domain analysis.* Domain analysis helps us identify suitable components, and structure the component library to aid retrieval.

Hall and Boldyreff [48] state that the reuse process consists of two quite separate phases, i.e., *a building-up activity (Design for Reuse and Reverse Engineering)* where reusable software is identified, classified and brought together into a library, and *a design activity (Design with Reuse)* where reusable software is retrieved and selected from the library according to system requirements and reused in building a new system. They also argue that for effective reuse, there is a need to integrate reuse with the traditional life-cycle of software development. They present one example of the integration of reuse with the traditional life cycle of software development.

Hooper and Chester [57] describe the process for reusing available reusable com-

13

ponents as follows: *classifying and storing components, locating components that fit specific needs, understanding retrieved components, adapting components where necessary, and integrating components into a system.*

Basili [3] treated maintenance as reuse-oriented software development and argued that "there are many reuse models, but the key issue is which process model is best suited to the maintenance problem at hand". He presented three maintenance process models: *a quick-fix model, an iterative-enhancement model, and a full-reuse model.* The models are reuse-oriented because all these models reuse the old system. The iterative-enhancement model supports reuse orientation more explicitly. The full-reuse process model reuses the appropriate requirements, design, and code from any earlier versions of the old system. Therefore, this model can be used as a reuse model for 'black box' reuse. In addition, Basili defined a reuse process as *identifying the reusable components, understanding them, modifying them, and integrating them into the process.*

Since Basili's full-reuse model starts with the requirements for the new system, reusing as much of the old system as feasible, and builds a new system using documents and components from the old system and from other systems available in the repository, the model presents reuse at higher levels of abstraction as well as the code level.

To develop software with reusable components, Hall and Boldyreff [48] identify two types of solution: finding a single component which will fully meet our requirements using the retrieval mechanisms, and finding some combinations of several components which could satisfy the requirements.

In summary, comparing the above reuse models, only Basili's reuse model considers the perspective of maintenance in software reuse. The drawback of the model is that it does not identify the activities of each phase. Hooper and Chester's model describes detailed activities of the reuse process together with two case studies. Hall and Boldyreff's model consists of identifying software for the reuse library, storing

14

and retrieving reusable software, and designing a new system with reusable software. The model includes 'Design for Reuse (DfR)' as well as 'Design with Reuse (DwR)'.

The process of reuse must be integrated into the overall development life cycle and maintenance model so that developers and maintainers can know the steps where they incorporate reuse. In addition, the incorporation of reuse into the life-cycle helps an organisation to accomplish systematic reuse.

In the next section, DwR will be described in terms of 'black box' reuse and 'white box' reuse as this research is focussed on 'Development with Reuse', not 'Development for Reuse'. In addition, a good starting point for undertaking reuse is to assess the maturity level of organisational reuse with respect to the Reuse Capability Model (RCM). The result of the assessment can be used for establishing the process of reuse applicable to each organisation. In the following section, RCM will also be described.

Development with Reuse (DwR)

Although the process of reuse consists of 'Development for Reuse (DfR)' and 'Development with Reuse (DwR)', this section investigates 'Development with Reuse (DwR)' because this research is concerned with Software Configuration Management (SCM) for a reuse library which supports a legacy system. The definition of 'Development with Reuse (DwR)' and methods required for successful DwR are described in this section. DwR is supported by 'black box' reuse and 'white box' reuse. Since there exist many synonyms concerning 'black box' reuse and 'white box' reuse, all synonyms including 'black box' reuse and 'white box' reuse are presented below.

'Design with Reuse (DwR)' has to do with finding the right components and glueing them together properly, whereas *'Design for Reuse (DfR)'* is responsible for making components which are readily retrievable (ie. describable) and easily glued together without much additional effort [15]. The term 'Design with Reuse' has the same meaning as 'Development with Reuse' and the term 'Design for Reuse' is equal

to 'Development for Reuse'.

In 'black box' reuse, a component is reused on an "as-it-is" basis, whereas in the case of 'white box' reuse, it should be modified before reuse. There is a useful distinction to be made between *'industrial reuse'* and *'research reuse'*. These are *'long term reuse'* and *'short term reuse'*, respectively, and there is a real difference between them. 'Industrial reuse' is equal to 'black box' reuse and 'research/exploratory programming reuse' is equal to 'white box' reuse [11].

Buckley [19] classifies reusable software into two variants. First, the software in the library itself can be rigidly controlled, particularly as it increases in value (fully tested, fully documented). In such cases, changes are formally proposed to a company *configuration control board (CCB)* and, when approved, all users (reusers) of that software are so notified. In a second variant, reusable software from a company library is provided on an "as-is" basis to any project requesting it. This is similar to software provided by a user group. Further modification and/or control of such software for use in a particular project is then delegated to the specific project using the software. If the modifications may be of use to another project, the revised software may also be placed in the company library.

Buckley's view is not very different in concept from 'white box' and 'black box' reuse. As he viewed software reuse in terms of Software Configuration Management (SCM) he stated that changes related to 'white box' reuse should be controlled by the CCB. The possibility of multiple versions in the reuse library was presented. Even if 'black box' reuse does not need to modify code, a library maintainer may modify the reusable component over time. Therefore, with respect to reusable software, change needs to be controlled in two respects, i.e., 'white box' reuse and 'black box' reuse.

In addition, there exists the distinction between *'total reuse'* and *'partial reuse'*. In the first case, a component is reused on an "as-it-is" basis, whereas in the second case, it may be subject to modifications. For this latter case, Design for Reuse (DfR) would also encompass *"Design for Modifiability"* [15]. 'Total reuse' and 'partial

16

reuse' can be mapped on to 'black box' reuse and 'white box' reuse, respectively.

There might be *'gray box'* [15] reuse between 'white box' reuse and 'black box' reuse, where it allows a few changes to reusable software components, e.g., renaming variables and changing parameters. Thus, whatever the type of reuse is, change control is required for changes to reusable components.

In order to be able to develop or design software using reusable components, there is a need to provide methods and support for the following: [11]

1. a software life-cycle incorporating reuse;

2. methods for adaptation and interconnection of reusable components;

3. support for checking the correct interfaces and coherence of composite systems;

4. a reuse economic model for estimating costs and benefits of reuse;

5. an information service centre about the reusable components, i.e. resources;

6. version/change and development histories— software component log books;

7. a means of capturing development histories that can be reused;

8. a means of identifying relationships between existing components.

Of all the methods and support identified above, most of the methods i.e., 1), 2), 3), 6), 7) and 8) can be mapped on to the issues of Software Configuration Management (SCM).

The selection of a suitable reuse method from the two methods (e.g. 'white box' reuse and 'black box' reuse) for 'Development with Reuse' should be decided by the Reuse Capability Model (RCM) [32] and the Software Engineering Institutes (SEI)' Capability Maturity Model (CMM) [84]; the former is described in the next section. The objective of the RCM is to assess the current level of maturity in software reuse and provide suggestions for a stepwise improvement. When an organisation first

introduces reuse, it has to assess its development process or maintenance process with respect to the SEI's CMM in order to build up the most applicable reuse model.

The Reuse Capability Model (RCM)

A reuse capability model provides users with a basis for understanding and improving an organisation's reuse capability. Reuse capability means the range of reuse results which an organisation can expect after reuse adoption.

Paulk, Curtis et al. [84] state that a capability model is a guide for selecting improvement strategies by determining current capabilities and identifying the issues most critical to improvement. A *Reuse Capability Model (RCM)* is a self-assessment and planning guide for improving an organisation's reuse capability which adheres to these concepts of technology adoption and improvement. Prieto-Diaz [91] argues that the problem is not the lack of technology for reuse. The problem arises when organisations implement a reuse programme using an independent collection of tools and techniques, or when an organisation focusses on the technical issues of reuse without adequately addressing the managerial issues. Blyskal and Hofkin [10] and Koltun and Hudson [66] have made initial attempts at reuse maturity models. Since reuse is performed in the context of a software development process, concepts from the SEI's CMM [84], have also been worked into the RCM where appropriate.

In this section, firstly, the reuse adoption process is explained in order to present how an organisation identifies its objectives for the reuse programme, assesses the current situation, selects a reuse adoption strategy and implements a reuse programme. Secondly, two components of the RCM are described such as an assessment model and an implementation model. Finally, two possible cases for reuse capability, i.e., a low reuse capability and a high reuse capability are addressed in terms of reuse proficiency, efficiency, and effectiveness.

18

The Reuse Adoption Process Davis' [32] reuse adoption process is based on the implementation model defined in Prieto-Diaz [91]. The reuse adoption process includes the following activities:

- **Initiate Reuse Programme Development.** This activity includes the identification of organisation objectives and reuse opportunities.

- **Define a Reuse Programme** This includes the definition of objectives for the reuse program, assessing the current situation with respect to reuse, setting reuse adoption goals, identification of constraints, and identification of alternative reuse adoption strategies.

- **Analyse Reuse Adoption Strategies.** This includes the assessment of risks associated with each alternative strategy. The purpose of this activity is to refine, evaluate, and select a reuse adoption strategy for implementing a reuse programme,

- **Develop a Reuse Action Plan** This includes the identification of the tasks, resources, and schedule for implementing the reuse programme according to the selected reuse adoption strategy.

- **Implement and Monitor a Reuse Programme** This includes monitoring progress against the plan and making any necessary adjustments. The RCM is used in the 'Define a Reuse Programme' activity as a means for the assessment of the current situation and in setting reuse adoption goals.

Components of the RCM Przybylinski, Fowler et al. [92] argue that to successfully adopt a new technology, the organisation must understand its present state of practice, be able to identify the desired state of practice, and develop a strategy that will successfully move the organisation toward the desired state. The RCM has two components: an assessment model and an implementation model. The assessment model consists of a set of critical success factors, defined in terms of goals, which

19

are used by an organisation to assess the present state of their reuse practice. Once an organisation completes the assessment, it selects the desired state and develops a strategy, which can be a very complex task. The implementation model helps organisations in prioritising the critical success factors by partitioning them into a set of stages [32].

Reuse Capability   Davis defines *'reuse capability'* as the range of expected results in *'reuse proficiency'*, *'efficiency'*, and *'effectiveness'* that can be achieved by an organisation's process. There are two possible cases for reuse capability: one resulting in a low reuse capability, the other in a high reuse capability. The low reuse capability case is characteristic of an ad hoc approach to reuse where the potential opportunities are not identified. This case results in a low reuse efficiency, proficiency and effectiveness. The second case is characteristic of systematic reuse [118] where the organisation identifies its potential opportunities, ensures the target set of opportunities falls within the potential, and has a process which ensures the target is met. Davis stresses that the aim of the model is to assist an organisation so that it can achieve the type of results of the high reuse capability case to be able to gain more *'reuse proficiency'*, consistently meet reuse targets *'efficiency'*, and maximise the payoff from *'reuse effectiveness'* [32].

In conclusion, before an organisation applies software reuse to a software development or maintenance process, it needs to identify the level of reuse capability. Then an organisation can choose a suitable method from the two methods (e.g. 'white box' reuse and 'black box' reuse). An organisation that has already standardised the process of software development and has a high reuse capability, can start with 'black box' reuse. On the contrary, some organisations where the degree of standardisation of the software development process is still low and which have a low reuse capability, may practise 'white box' reuse first and then can introduce 'black box' reuse when they reach the mature level of reuse capability.

2.1.4 Reusable Software Library

Although there have been simple collections of reusable components and small reuse library systems over the last decade, several large reuse libraries have been produced recently and are now available via the World Wide Web (WWW). These libraries are usually funded by large organisations such as DoD, NASA, ESPRIT and universities and access is free. The libraries are described below, and a comparison of these libraries is made in the next chapter.

AdaBasis AdaBasis [103] is a repository of free Ada Software that is constructed in an easy-to-use way and allows flexible access and effective searching. The software in this repository is based mainly on the "PAL (Public Ada Library)" and is still growing. It is presented in a hierarchical manner and separated into different application domains. It also has a searching facility for some domains.

ELSA ELSA (Electronic Library Services & Applications) [114] that has recently been renamed "Software Market", is a NASA funded service provided by MountainNet, providing access to a large selection of high quality software examined for integrity and compatibility. The ELSA project has focussed on introducing and supporting common, effective approaches to designing, building, and maintaining software systems by using existing software assets stored in a specialised library or repository. ELSA provides a customer-driven environment employing an advanced library management mechanism, MORE (Multimedia Oriented Repository Environment).

ASSET ASSET (Asset Source for Software Engineering Technology) [95] is sponsored by the Advanced Research Projects Agency (ARPA) organised under the STARS (http://www.stars.ballston.unisysgsg.com/index.html) programme. The ASSET Reuse Library serves as a national resource for the advancement of software reuse across the Department of Defence (DoD). ASSET's mission is to provide a

ing. In particular, RSE (Reusable Software Engineering) requires more reliable and reusable software components to be produced through the process of system modelling. In terms of software reuse, the activity of system modelling is associated with one of the activities of a 'Development with Reuse (DwR)' process as the DwR process includes an activity of building a new system using reusable components.

The Jasmine system [78] is one of the first systems to explicitly use system models for the representation of system structure. Marzullo and Wiebe [78] proposed that four categories of information should be included in a system model: relations between system components, version information, construction rules for system building, and verification rules.

There exist different approaches to system modelling. Perry [86] classifies *Software Interconnection Models* into three classes, i.e., *the Unit/Basic Interconnection Model, the Syntactic/Structural Interconnection Model and the Semantic Interconnection Model*. The Basic Interconnection Model represents the relationship between modules or files of a system. 'Make' that has been widely used on the Unix system is an example of this model.

This section describes and compares most of the Module Interconnection Languages (MILs) that are used for Syntactic Interconnection Modelling, and the Component Description Languages that support Semantic Interconnection Modelling.

Syntactic Interconnection Model

This interconnection model specifies the structure of a system, not the behaviour of a system. *Module Interconnection Languages (MILs)* such as *MIL75, Thomas' MIL, Cooprider's MIL, INTERCOL, SySL and PCL* are examples of this type of model. MILs are essential tools in the development of large software systems. Current software development practice follows the principle of the recursive decomposition of larger problems that can be grasped, understood, and handled effectively by independent development teams. After designing and coding their respective

23

subsystems successfully, teams are faced with more difficult issues; how to integrate independently developed subsystems or modules into the originally planned complete system. MILs provide formal grammar constructs for describing the whole structure of a system and for deciding the various module interconnection specifications required for its complete integration. While the major benefits of a MIL may seem to be during the system design phase of the software life-cycle, the actual benefits are during system integration, evolution and maintenance. This is because the system specification described by the MIL is a written description of the system design which must be followed for every system version that is built [89].

The modelling of software architectures using the first Module Interconnection Language (MIL), MIL75, was introduced in 1976 by DeRemer and Kron [33] for *'programming-in-the-large (PIL)'*. Subsequently MILs have found importance in software reuse, as a means of interconnecting components. After a system structure is specified, it may be coded using a MIL to be checked and verified for completeness and consistencies. MIL code should be maintained during implementation and then used for high-level maintenance during system operation and evolution [89]. The following paragraphs outline the four MILs, i.e., MIL75, Thomas' MIL, Cooprider's MIL and INTERCOL, and more improved Syntactic Interconnection Languages such as SySL and PCL in chronological order.

MIL75 MIL75 [33] is based on the decomposition principle fundamental to structured design and represents a system in the form of an inverted tree structure. MIL75 consists of three sets of items that are required to describe the structure of a system:

1. *Resources*: Atomic elements that represent abstractions of programming constructs within a program and are available for reference to other modules (e.g., variables, types, arrays, functions, etc.).

24

2. *Modules*: Programming units made by resources and other programming constructs that perform a specified function or task.

3. *Systems*: Groups of hierarchically organised modules that communicate via resources to perform defined functions.

The main contribution of MIL75 is in providing the designer with some means of finding incorrect design decisions before implementation begins. The rigidity of MIL75 is its drawback, caused by its attachment to the inverted tree structure. Another weakness of MIL75 is its lack of support for the 'specification of the function of the modules' [89].

Thomas' MIL Thomas' MIL [104] was proposed to enhance flexibility of module interconnection and to reduce constraints to a particular system structure that exist in MIL75. The "universe of discourse" of Thomas's MIL is *names* that are classified into four classes: *Resources, Modules, Nodes, and Subsystems*. The definitions of resources and modules are almost identical to the ones given in MIL75.

Nodes are descriptive units (in MIL code) that construct environments for the modules by combining resources with modules. Nodes are the basic entity for 'programming-in-the-large (PIL)'.

Subsystems are graphs(directed) of nodes and the edges connecting them with one node (the 'distinguished node') providing a characterisation of the subsystem.

The pay-off of Thomas' MIL accrues during maintenance when individual modules can be added without requiring full recompilation of the system. The drawback of Thomas' MIL is the binding of the interconnection to the compile/link paradigm. Thomas' MIL was only a discussion of a possible MIL processor and it was not implemented. However, his work is valuable as it provided some basic proofs on MIL structures and proposed some ideas for future MILs [89].

Cooprider's MIL The objective of Cooprider's MIL [27] is to propose a system that would bridge the gap between software design and software construction. He developed a representation for software systems that integrates a MIL, a version control facility, and a software construction facility. This MIL provides three levels of notation. The highest most abstract level specifies the interconnection between subsystems or modules. The intermediate level presents instantiations of system versions according to the structures of interconnection. The lowest most concrete level describes operations for actual system construction.

In comparison with the two previous MILs, Cooprider's MIL can be regarded as an extended MIL that supports system construction as well as system design. The weakness of the construction system is that the database has no information on the various versions. Several parts of this MIL have been implemented and the implemented components have been tested in a laboratory environment [89].

INTERCOL INTERCOL [106] introduces a new approach to the representation of modules in the system, since its description is a sequence of module and system families followed by a set of compositions. A member of a module family is a version of a module and a member of a system family is a version of a system. Tichy's work at the software development environment level has three objectives:

1. *A Module Interconnection Language* for representing multiple versions and configurations written in several programming languages.

2. *An Interface Control System* that automatically checks interface consistency between separately developed software components.

3. *A Version Control System* similar to the version control system of Cooprider's MIL but with the advantage that the system decides which version of which component should be used to form a particular version of a particular configuration without relying on a detailed set of construction commands given by the designer as in Cooprider's MIL.

26

INTERCOL's increase in the detail of the interface description is needed for more effective type checking but it forces the system developer into premature decisions about module implementation. INTERCOL is embedded in a Software Development Control Facility (SDCF) which is an interactive system that controls a database for software development. The payoff of Tichy's SDCF occurs in controlling the evolutionary process of a software system. The approach of system design by 'evolving prototypes' would be the ideal approach to use in this SDCF [89].

MILs were designed to specify not the behaviour of a system but the structure of a system. DeRemer and Kron stated that a MIL did not support functional descriptions of software components. In terms of software reuse, this characteristic of the MIL is a kind of weak point since the functional description of a module is very important for software reuse.

SySL The SySL (System Structure Language) system [105] was used as the basis for an Esprit research project called SESE. The aim of the project is to integrate both process management and configuration management with SySL. SySL allows developers to document the components that make up the system, record the relationships between the different versions of these components and automatically generate the build files necessary to create a version of the system.

SySL provides a notation for modelling all associated information about a system, including software, hardware and documentation. Therefore, the objective of SySL is to generalise the notation and allow the structure of all types of systems which can be represented in a database system to be described in the language. SySL provides the following basic features:

- SySL provides facilities for describing systems or families of systems at various levels of detail and abstraction.

- SySL provides facilities for putting constraints on particular combinations of items and item attributes.

27

- The language allows the description of any structured system whether it is hardware, software, documentation, etc.

PCL Sommerville and Dean [101] describe a configuration language called PCL (Proteus Configuration Language) which has been designed to represent the architecture of multiple versions of evolving systems. The features of PCL include the modelling of variability between the different versions of a system, support for object-oriented models, support for specifying relationships between different parts of the model, and version identification and binding through attributes. The objective of PCL was to develop a means of representing the architecture of all the different system versions in a single model. After studying several different application domains, they identified several requirements for an ideal configuration language as follows:

1. *Integrated system modelling.* The language for the architectural modelling should model all information about software, hardware and documentation structures.

2. *Multiple structural views.* The language must provide different structural views of an entity and system to be built such as *an interface view, an entity view, a system view and a component view.*

3. *Variability expression.* The language must provide facilities to represent different versions of a system and to show clearly the difference between versions.

4. *Object-Oriented modelling.* The language must support object-oriented modelling in order to develop object-oriented extensions to existing design methods.

5. *User tailorability.* The language must be extensible so that it can be tailored to interface easily with other design methods.

Sommerville and Dean [101] compared PCL with other module interconnection languages with respect to the requirements described above. Their comparison shows

that PCL is the best Module Interconnection Language, whereas SySL is relatively better than the other MILs, i.e., MIL75, Thomas' MIL, Cooprider's MIL, INTER-COL and NuMIL [96]. In order to model hardware, software and documentation entities and the relationships between them at different levels of abstraction, they established 6 basic types of entity which PCL should include: *family entities, version descriptor entities, tool entities, classification definitions, relation definitions, and attributes type definitions.*

Summary To summarise, MIL75 was the first MIL and a language for PIL. It established the basic ideas and concepts of a MIL. Thomas developed a module interconnection notation and discussed a possible module interconnection processor. Cooprider expanded the basic ideas of MIL75 to introduce a version control facility and a software control facility. Tichy developed INTERCOL that integrates some of Cooprider's features with control of system families, and with independent compilation of modules and type checking. SySL allows developers to document and record the components of the system and the interrelationships between them. Finally, PCL represents the architecture of multiple versions of evolving systems and supports the modelling of variability between the versions of a system.

Semantic Interconnection Model: Component Description Languages

Semantic Interconnection Modelling is supported by a Component Description Language. *A Component Description Language* is a structured language used to capture the essential attributes of components in a specific domain. Perry [86] classifies Inscape [86, 87] and LIL [45] as examples of a Semantic Interconnection Model. This model defined by Perry is closely related to two *reusable component models*, the 3C model [109, 110] and the REBOOT component model [64]. The 3C model uses a prescriptive approach to component modelling that defines the attributes of a component that Component Description Languages for reuse should represent, whereas the REBOOT model describes components that have already been built and it

29

can only be used if a component fits within the given framework of facets [121]. Thus, Component Description Languages implemented using the 3C model, such as OBJ [44], LIL [45], RESOLVE [53, 52], etc. are described below.

OBJ Languages   Goguen [44, 46, 47] developed the OBJ series of languages that support parameterised programming. Parameterised programming is a powerful technique for software reuse. Modules through this technique are parameterised over very general interfaces that describe what properties of an environment are required for the module to work correctly. The objective of parameterised programming is to maximise program reuse by storing programs in as general a form as possible. A reuser can build a new program module from an old one just by instantiating one or more parameters. The OBJ programming language consists of three basic building blocks: *theories* which declare the properties of program modules and interfaces as a whole; *views* which connect theories with program modules; and *module expressions* which are a kind of general structured program transformation that builds new modules by modifying and combining existing modules. OBJ is supported by a simple logical system that is equational logic; moreover, these high level descriptions of what a program does actually are the program that a user can execute. Therefore, OBJ is a "logic programming language" such as Prolog and LISP [44].

LIL   Goguen [45] presents a *Library Interconnection Language (LIL)* that extends theories and views introduced in OBJ and features modest use of semantics in order to assemble large programs from existing entities. He states that LIL consists of a part like Ada's specification part, plus commands for interconnecting components to form systems. LIL syntax is similar to Ada, but this can be easily changed for use with other languages. The *package* is a basic LIL entity and generalises Ada's specification part in two main ways: *axioms* can be used for the operations the package declares; and the package may have (zero or more) *versions*, which are packages that realize the behaviour it describes. Therefore, LIL can support both semantic specification and version control.

LIL encourages the reuse of code and programming experience by binding *theories* to software components using *views, generic entities and a distinction between vertical and horizontal software composition* [121]. Theories integrate semantic descriptions with software components by providing axioms, either formal or informal. Views present semantically correct bindings at software interfaces and interconnections. *Horizontal composition* involves imposing structure at a given level of abstraction whereas *vertical composition* is associated with moving between different levels of abstraction [45].

CDL   CDL [43] was developed as part of the Alvey Eclipse programme [14]. CDL supports the functionalities to describe the interfaces of a reusable component. Like other component description languages such as OBJ and LIL, CDL belongs to an object-structured paradigm. CDL components consist of two parts: an *interface* that describes the facilities exported by the component and its relationship with other components; and a *body* which presents the implementation of the ideas expressed in the interface. Both the interface and body use an Ada like syntax. CDL belongs to a language of the design level, so its specification is transformed to the implementation level [121].

CIDER   CIDER [122, 14] is an object oriented component description language. The properties for the component interface of CIDER are paramount and are composed of the interfaces of a data type and the interfaces for operations of that data type. The language provides a powerful syntax. A parameterisation mechanism is flexible as it supports inheritance, importation and instantiation. Component interface descriptions can be extended through the definition of operations for those descriptions.

LILEANNA   LILEANNA (LIL Extended with Anna:Annotated Ada) is a language for formally specifying and generating Ada packages [111, 112]. This combines

31

the facilities of Goguen's LIL and those of ANNA [77, 76]. LILEANNA extends Ada by introducing LIL's two entities: theories and views, and enhancing the Ada package specification. Using semantics specified formally or informally, a LILEANNA package represents a template for actual Ada package specifications. It is used as the common base for families of implementations and for version control. As in OBJ, LIL programs are structured both horizontally and vertically using inheritance and aggregation. Inheritance is used to express relationships between content and concept that consist of code and type information. The payoff of LILEANNA is the ability to instantiate and manipulate existing packages [121]. Another benefit is its applicability to support a "megaprogramming" software development paradigm.

RESOLVE RESOLVE (REusable SOftware Language with Verifiability and Efficiency) [53, 52] is primarily a research vehicle that has allowed users to understand better how to synthesise and formally express many important ideas about software components. It has been used to build software components and applications in 'real' languages such as Ada and C++. RESOLVE has the following three meanings:

- a conceptual *framework* to guide thinking about a component-based approach to system development;

- a specific *language* to provide an easy description of components and systems within that framework; and

- a general *discipline* for using the language to design high-quality software components and systems [119].

RESOLVE uses a mathematical model based on a formal specification approach. This language enables a designer: to model a component's types using existing theories; to provide pre- and post-conditions for the operations of components; to define the conventions to be used for the component's implementation; and to specify the correspondence between the model and implementation [121]. RESOLVE can be

32

used stand-alone, and also subsequently translated into implementation languages such as C++ or Ada [98].

Conclusions

In conclusion, SySL is relatively better than other traditional MILs as this language can support the modelling of software, hardware and documentation. SySL is still an experimental system as it was not applied in an industrial environment. PCL can be considered the best Module Interconnection Language with respect to several requirements for an ideal configuration language as shown in Section 2.1.5. However, MILs have a weak point in that they do not support a functional description of a module, which is very important for software reuse. For this reason, Component Description Languages (e.g., OBJs, LIL and RESOLVE) are more suitable for constructing a system using reusable components because they can support Semantic Interconnection Modelling which specifies both the behaviour and structure of a system. Of Component Description Languages, RESOLVE has more strengths than other Ada based languages such as OBJ, LIL, LILEANNA and CDL as it can be used stand-alone and subsequently translated into implementation languages like C++ or Ada. In addition, RESOLVE is based heavily on Goguen's OBJ and LIL, as well as MILs.

2.1.6 Relationship between Software Reuse and Maintenance

Hall and Boldyreff [48] argue that to view maintenance as reuse is a misuse of the term and that this view is unhelpful. However, they state that some of the practices (i.e., modularity and consistency) and mechanisms (i.e., flexible module connection methods and the ability to regression test) necessary to support maintenance are also necessary to support reuse.

Basili [3] treats maintenance as a reuse-oriented development process. He presents

three maintenance process models: the quick-fix model, the iterative-enhancement model, and the full-reuse model. A full-reuse process model starts with the requirements analysis and design of the new system and reuses the appropriate requirements, design, and code from the old system. A full-reuse process model can be related to 'black box' reuse as this model reuses specifications as well as software code. Thus, Basili's full-reuse model is most appropriate to a maintenance model for an organisation that has a high reuse capability in terms of reuse proficiency, efficiency and effectiveness.

To successfully introduce reuse into the software development process, we should integrate the reuse library into existing software tools and CASE environments [79]. In addition, the process of reuse must be an integrated part of a maintenance environment so that reusers and maintainers can perform systematic reuse and enhance the effectiveness of reuse. Thus, a maintenance model must incorporate the activities of the reuse process in the stages of the maintenance model.

Software maintenance, and particularly re-engineering, are concerned with software reuse. The Object-Oriented Technique (OOT) that supports 'Design for Reuse (DfR)' can create maintainable, reusable specifications and code. Although legacy systems were not developed using an Object-Oriented method, they might contain potentially reusable components. Re-engineering technology enables a maintainer to create/extract reusable assets from existing system components, thereby enhancing the productivity and quality of software maintenance. Maintenance and reuse are very cooperative with each other as re-engineering/reverse engineering can produce reusable components using an existing system, and reuse can support a maintenance process through the reuse process based on 'Design for Reuse (DfR)' and 'Design/Development with Reuse (DwR)'.

2.2 Software Configuration Management (SCM)

2.2.1 Introduction

Bersoff [8] argues that *Software Configuration Management (SCM)* is one of the disciplines, with both management and technical dimensions, employed to attain and maintain product integrity. He identified the "supporting" disciplines and "doing" disciplines as requisite disciplines for attaining and maintaining product integrity, and viewed SCM as the "supporting discipline/the product insurance discipline" such as *Quality Assurance, Verification and Validation, and Test and Evaluation.*

Babich [2] states that the technique of coordinating software development is required to minimise the degree of confusion on any team project and that SCM is the method for identifying, organising, and controlling modifications to the software being built by a programming team, in order to maximise productivity by minimising mistakes. He presents three typical problems that serve as examples of the need for SCM: *the double maintenance problem; the shared data problem; and the simultaneous update problem.*

Buckle [18] defines SCM as a collection of techniques that improve the quality of the software product, reduce its life-cycle costs, and improve the management function for the development process. He presents four basic concepts of SCM as follows: *identification, control, status accounting and verification.* IEEE standard 729-1983 [61] defines SCM as four classic operational aspects: *identification, control, status accounting, and audit and review.*

Comparing with the first two views of SCM, Bersoff's view focusses on the managerial aspect of SCM, whose aim is to enhance product integrity, i.e., software quality. On the contrary, Babich's view concentrates on the technical aspect of SCM, whose aim is to maximise team productivity through *change control and version control.*

SCM is a key element of the process of software maintenance as well as devel-

opment. SCM was not regarded as an important issue in the field of software engineering until the early 1980's, because the software community usually placed higher emphasis on producing the product on time rather than on software quality. They thought SCM was more related to managerial issues rather than technical issues. Thus, as SCM was considered the solution to management problems, it was not taken into account as a key issue by the software engineering community that was looking for solutions focussed on technical problems.

However, SCM has become a more well founded part of software engineering in the last decade. For instance, SCM is an important component of several levels in the software process maturity model [58]. In addition, SCM related standards have been developed and improved by IEEE [61]. Commercial systems have introduced new SCM concepts(functionalities). These new concepts that can be classified into *standalone SCM tools, environment frameworks with SCM capabilities and CASE tools with multi-user support*, are steps beyond the initial *check-out/check-in model.*

In this section, firstly, the developments of SCM are described. Secondly, major activities and models of SCM are presented. Finally, automation of SCM is addressed, followed by SCM problems within a maintenance environment.

2.2.2 Evolution of Software Configuration Management

Dart [31] argues that the past and present situation concerning SCM systems should be investigated in order to identify future SCM challenges. The past focused on in-house SCM solutions whereas the present concentrates on any third-party SCM solutions. The SCM future is associated with technological, process-oriented, political, standardisation and managerial challenges. One way to address these challenges is to define an SCM services model.

The Past

In the past, along with a few automated facilities such as SCCS and Make, most of the SCM solutions involved manual procedures and policies. In general, the past brought about a good understanding of version control, compiling code, tracking and dealing with bugs, and a realisation that simple version control was not the complete solution to SCM needs.

The Present

The present is characterised by a better understanding of SCM technology, such as the work performed by the Software Engineering Institute (SEI) at Carnegie Mellon University and a recognition gained from practice and experience within the software engineering community. The SEI has identified process maturity levels for an organisation and a few key practices central to carrying out SCM [84]. As organisations begin to more formally define their process models and evaluate their process maturity levels using the SEI process maturity levels, it becomes clear that SCM capabilities play a major part in attaining a higher process maturity level.

In particular, after a survey of tools and environments, Dart extracted a set of 15 SCM concepts [30] that capture the essence of automated support for SCM. These concepts are as follows: *Repository, Distributed Component, Context Management, Contract, Change Request, Life-cycle Model, Change Set, System Modelling, Subsystem, Object Pool, Attribution, Consistency Maintenance, Workspace, Transparent View,* and *Transaction.* Repository, Change Request, Change Set, System Modelling and Object Pool are explained in brief below as they are concerned with this research.

Repository captures SCM information and stores versions of files as frozen objects, as in RCS [108].

Change Request assists in treating the process of change to configurations and

keeping an audit trail of the changes, as in LIFESPAN [120].

Change Set addresses a logical change to a product and a means of creating any version of a configuration that is not necessarily dependent on the latest version of that configuration, as in ADC [1].

System Modelling abstracts the notion of a configuration from an instance of it and by fully describing the configuration, and assists in maintaining the configuration's integrity, as in Jasmine [78].

Object Pool optimises the need to regenerate objects and maximises the amount of sharing of derived objects, as in DSEE [73].

These concepts also provide a base that enables people to discuss automated SCM support. While many concepts are automated in SCM systems, no single SCM system provides all the concepts to meet all kinds of users' needs [31].

The Future

Technological Issues New requirements of SCM functionalities and better implementations of these requirements are needed for the future of SCM.

Process-Oriented Issues An SCM process definition and automated support are required to implement an SCM system. This issue requires a detailed definition of SCM processes; an understanding of how much control will be enforced compared to how much guidance will be given by the process manager; adequate implementations; and monitoring of how well the process is followed and where improvements can be made.

Managerial Issues To solve the SCM problems in an organisation, it is necessary to obtain better management support; that is, give management an understanding

of the complexity of the solution and hence the costs and tradeoffs. Management must be prepared to make the "buy versus build" decision in examining possible SCM solutions. Przybylinski et al. [92] suggests that it is necessary to deal with the technology transition issues of introducing SCM technology into an organisation, such as convincing people to use SCM, after management evaluates SCM systems and their capabilities as part of finding a solution.

Political Issues   It appears that future government contractors in the U.S. will be required to use certain SCM tools in order to make a contract. For instance, a contractor would have to be a level 3 organisation, where that level is based on the SEI's Capability Maturity Model.

Standardisation Issues   SCM is being recognised as a key factor in environment framework standards.

An SCM Services Model

Dart [31] introduced an SCM services model as a way of starting to address some of the above five issues for the SCM future. An SCM services model is a conceptual framework that builds a set of 'well-defined' services related to SCM functionalities. The services should be 'well-defined' so that their semantics, interfaces, and other properties can be understood in order to be included in the framework model. The services represent a combination of several viewpoints: end user, environment builder and tool integrator. The end user chooses capabilities for the SCM system, the environment builder provides tailoring features for the end user, and the tool integrator needs to mix and match existing capabilities and devise an SCM solution in a cost effective way.

SCM provides a good solution to the software engineering problem and it should not be viewed separately from other software engineering problems and solutions.

SCM capabilities are the groundwork of any software development environment. Good SCM support makes for a good environment whereas bad SCM support makes an environment unusable. An SCM solution is a microcosm of all issues affecting an environment, including technology transition, user requirements, roles, integration, databases, SCM technology, process modelling, education and training of users, and managerial and organisational decisions [31]. In this respect, SCM capabilities help software reuse to be introduced into an organisation successfully. To be able to implement *systematic reuse*, not *ad-hoc reuse*, an organisation should integrate software reuse with a software development/maintenance environment that is supported by SCM.

2.2.3   Software Configuration Management Activities

Bersoff [8] identified four basic activities of SCM as *configuration identification, configuration control, configuration auditing and configuration status accounting*. These basic activities have been effectively used as a basis for extracting many functionalities/concepts of SCM from the process of software development.

Dart [30] states that SCM activities of the operational level for SCM systems comprise *manufacture, process management* and *team work* as well as classical functionalities such as *identification, control, status accounting, audit and review*. Manufacture manages the building of the product in a cost effective way. Process management ensures the conformity of the organisation's standards, policies and life-cycle model. Team work controls the work and communication between multiple users on a system. Some key operations associated with the above activities are described in more detail in the sections that follow.

Identification

Bersoff [8, 6] defines SCM as the management discipline of *identifying* the proposed or implemented configuration of a system at discrete points in time for purposes of systematically *controlling* changes to this configuration and *assuring* the integrity, accountability, visibility, reproducibility, project coordination and traceability of this configuration throughout the system life cycle. This definition shows that the necessary first step in SCM is the identification of the software configuration at discrete points in time.

IEEE standard 729-1983 [61] defines configuration identification as a scheme reflecting the structure of the product, identifying components and their type, thereby making them unique and accessible in some form. For instance, this addresses the question, "What version of the file is this?"

The activity of identification enables the representation of a software system in a way which explicitly identifies the structures and components of the product, and relationships between components. In addition, this enhances the visibility of a product and permits the software to be seen well by anyone who can access the software components. It also improves traceability, i.e., the ability to link *Software Configuration Items (SCIs)* to each other. Ben-Menachem [6] presents the rules governing the process of *Software configuration identification (SCI)* as follows:

1. SCI defines the 'granularity' of SCM. Granularity implies the 'size of the grains' to be managed using an SCM system.

2. SCI defines what needs to be seen by all those who have to ascertain the status of the project.

3. SCI ensures that the chosen identification scheme reflects the structure of the product, the project and the organisation.

4. The identification process should always proceed together with a process of

labelling the item with a unique label.

5. Note that the size of the 'grains' can never be consistent.

6. SCI is a critical SCM task.

Configuration Control

IEEE standard 729-1983 [61] defines configuration control as controlling the release of a product and changes to it throughout the life-cycle by having controls in place that ensure consistent software via the establishment of the baseline of a product. For example, this is concerned with the question, "What changes have been made to the latest version of this product?"

This activity establishes the change control procedures to initiate, evaluate, approve and implement changes to a baseline. Ben-Menachem [6] states that once an item is tied into a baseline, changes are made only via a formalised process called the CCB (Change Control Board). The CCB is the committee whose purpose is the control of changes. It always consists of at least the following: a representative of users/client, a configuration manager, a project manager and an SQA (Software Quality Assurance) manager. The roles of CCB are as follows: define the information needs of the CCB; decide if the change request should be implemented; monitor the implementation of the change request; and verify the quality of new software components produced by change requests. The first responsibility of the CCB is analysis of incoming change requests. The CCB must analyse technically the implementability and desirability of the requested change. A disapproved change request should be returned to the originator. Such returns will be accompanied by sufficient reasons for disapproval. In addition, the CCB is also responsible for *release management*. The release of software and documentation should be controlled in order to ensure that only the correct versions of all components are used.

Buckley [19] describes the configuration control process as follows: firstly, iden-

tifying the problem, secondly, determining the corrective action, and finally, implementing the change. Implementing configuration control of software requires the use of a controlled repository, i.e., a configuration management library which controls access and supports strict check-out and check-in procedures.

Change control documents are used to issue and record changes to baselined items such as software and documents. The forms used for configuration control are *a Change Request(CR) Form, a Change Proposal Form/Engineering Change Proposal (ECP), a Software Incident Report (SIR), a Change Approval Form, a Document Tracking Form (DTF), a Software Change Notice (SCN), and a Software Patch Form.*

Status Accounting

IEEE standard 729-1983 [61] defines configuration status accounting as recording and reporting the status of components and change requests, and collecting vital statistics about components in the product. For instance, this addresses the question, "How many files were affected by fixing this one bug?" Thus, status accounting aims at meeting the following questions: what happened?; when did it happen?; why were the changes made?; which items were affected; and who authorised and made the changes?

Whitgift [120] argues that the objective of configuration status accounting is to enhance the visibility and traceability of Software Configuration Items (SCIs) by recording and reporting the status of all items and change requests. Although different people need different SCM information at different times and in different forms, Ben-Menachem [6] presents a minimal wish list of reports as follows: transaction log, change log, item 'delta' report, resource usage, stock status (i.e., status of items), changes in progress, and deviations agreed upon.

Configuration Audits

IEEE standard 729-1983 [61] defines configuration audit as validating the completeness of a product and maintaining consistency between the components by ensuring that the product is a well-defined collection of components. For example, this is associated with the question, "Are the correct versions of files used for this current release?"

Ben-Menachem [6] states that configuration auditing is best performed by an external auditor because a very high objectivity is required when auditing a critical management function. The more important the system is, the more this independence is critical. Of all SCM functions, change control always is the most important thing to audit.

Configuration auditing can be equally considered V&V (Verification and Validation). *Verification* ensures that Software Configuration Items (SCIs) conform to the specification of the baselines of previous phases. *Validation* involves checking that SCIs specified in the baseline meets end user's requirements. Thus, an organisation that has already performed V&V as a means of *Software Quality Assurance (SQA)*, can replace configuration auditing with the V&V activities.

The validation of maintenance includes regression testing, which ensures the absence of unanticipated side-effects in other components which have not been modified. It also involves auditing whether software follows design principles, coding standards and other quality standards [22].

Version Management

Clemn [24] describes version management as *History Management* that provides the user with information about the historical development of existing software components to help guide future development. History management is usually provided in one of the following ways: *internal annotations* in the form of comments in the

software component, textual *log message* attributes associated with a new version when the version is completed, and special *modification request* objects associated with a new version when the version is initiated.

Version management requires efficient disk storage of the actual configuration items, i.e., the archives. Ben-Menachem [6] classifies methods for delta storage as *forward delta and reverse delta storage*. The method of forward delta storage maintains a complete copy of the original file, as first provided to the configuration manager. The delta information is appended to the file with every 'check-in' operation. Every update(check-in) operation requires a creation of the latest revision in order to generate deltas for a new revision. The major problem with this method is that more revisions require more retrieval time. The method of reverse delta storage stores a full copy of the latest revision. The latest revision that is most often used is always available without any creation process. As this method does not need any computation, the process of check-in for a new revision is much faster. Thus, the reverse delta storage method has been adopted by most tools for version control because of its inherent efficiency.

During software development and maintenance, most of the Software Configuration Items (SCIs) evolve until they meet user's requirements. These changes of SCIs create a new revision or a variant of the configuration item. The procedure to modify a configuration item is to check out the item from the software library(repository), change the item and then check the new version back into the library. Whitgift [120] describes the differences between a revision, a variant and a version as follows:

Revision An instance of a module(item or workfile). One item version is a revision of another if it was created by modifying the earlier version which it supersedes. During the maintenance phase of a project, revisions are needed to correct, perfect, adapt and improve the software, as described in Section 2.3.1.

45

Variant An alternate form of a module. Variants allow one item to meet conflicting (i.e., different but related) requirements at the same time: temporary variants allow parallel development and will eventually be merged; permanent variants will not be merged but enable the item to meet different functional requirements (e.g., various platforms, various user requirements and the particular requirements of testing and debugging) and will therefore exist as a series of revisions.

Version An instance of a (whole) system. Both variants and revisions are called an *item version*.

In general, a version is considered a general term that includes a revision and a variant. Here we need to clarify the notions of terms such as configuration control, change control and version control. In my view, configuration control is treated as an equivalent term to change control. However, as configuration control focuses on an activity that establishes the change control procedures to initiate, evaluate, approve and implement changes to a baseline, it is concerned with organisational issues and managerial-oriented aspects. On the contrary, since version control focuses on the *evolution (i.e., revision)* and *revolution (i.e., variation)* of configuration items within the software library(repository) over the software life-cycle, it is associated with technological issues of SCM and belongs to a micro level of SCM activities. In general, the activity of version control can be a subset of the configuration control activity. Configuration control and version management activities are central for this research work as these activities provide the reuser and maintainer with a means of controlling the evolution of components within the reusable software library and existing system.

2.2.4 SCM Models

Feiler [38] classifies the SCM models into four categories based on a set of 15 concepts described in Section 2.2.2. The four models, the *check-out/check-in model,*

composition model, long transaction model and *change set model,* are described in brief below.

The Check-Out/Check-In Model

The check-out/check-in model supports SCM functions as exemplified by Unix *SCCS* [93] or *RCS* [108] and *make* [40]. The tools of this model consist of two relatively independent tools: *a repository tool* and *a build tool.* The repository tool stores versions of files and provides mechanisms for creating new versions. The build tool generates automatically *derived files* such as object and executable code, through a description of the components that make up a system. The capabilities of the check-out/check-in model depend upon maintaining a version history of files, and upon controlling their concurrent modification using *locking, version branching and merging* facilities as follows:

- Revision: evolution of a sequential version history.

- Version branch (variation): version sequences that have a particular version in an existing branch as their starting point, but evolve independently.

- Merging: combination of two versions from different branches into a new version in one of the branches.

The latest version of a branch can be checked out for modification and the branch is locked. Branch locking ensures that only one person at a time is in the process of creating a new version for a specific branch. When the updated file is checked in, a new version is added to the branch and the branch is unlocked. SCM systems primarily supporting this model focus on controlling the repository.

The Composition Model

The composition model focusses on improving creation of configurations and management of their history. Developers deal with configurations by repeatedly composing a system from its components and by selecting the appropriate version for each component. A particular configuration of the composition model consists of a *system model* and *version selection rules (configuration thread)*. A system model contains all the components that make up a system. Version selection rules show which version should be chosen for each component to make up a system configuration. The selection rules are applied to a system model through the process of version selection that *binds* a component to a version.

These two processes of composition and selection can be graphically visualised as an AND/OR graph [107]. The first process is the integration of components into *a composite (an AND node)*. The second process is the selection of a suitable version for each of the *composite elements (an OR node)*. The structure of a system is captured in a system model. The system model combines configuration support, system build tools, and language systems so that the SCM system supporting the composition model can support management of derived objects and checking of interfaces between components and subsystems. The composition model can be supported by Module Interconnection Languages (MILs) described in Section 2.1.5. Thus, the composition model that consists of a system model and version selection rules can be equally applied to the process of software reuse.

The Long Transaction Model

The long transaction model supports the evolution of whole systems and coordinates the change of systems by development teams. In this model, developers primarily work using versions of configurations. Developers first select the version of the system configuration (i.e., *configuration version*), then consider the system structure. This view of version selection is contrary to that of the composition model, where

the developer first decides the system structure (i.e., *system model*) and then selects the versions of components to use. Once the version of the system configuration is selected, the appropriate component versions are implicitly inferred from the configuration [38].

This model consists of two concepts: a *workspace*, and a *concurrency control scheme*. A workspace, representing the working context, provides local memory, i.e., data storage visible only within the scope of the workspace. It provides stable workspaces with control over isolation from external change, scopes of visibility for changes, and coordination of concurrent change activity [39]. The long transaction model does not directly support composition, but does support the evolution of subsystems based on a decomposition of the system structure. After the versions of the system configuration are verified and validated, these system configurations can make up a *system/product family* that can be used as independent development paths treated as *system versions or variants*.

The Change Set Model

The change set model supports management of logical changes to system configurations. The change set is the set of differences between two configuration versions. This set of differences is the collection of deltas of those components that have been modified between the two configuration versions. In this model, configurations consist of a baseline and a set of change sets. In other words, a configuration version is mapped to one baseline. This view of SCM can be termed as *change-oriented* SCM due to its focus on logical changes [74]. Change-oriented SCM differs from the *version-oriented* SCM related to the other three SCM models which focus on versioning of components and configurations.

The change set contains all changes to all files in the configuration along with the reason for changes and details of who made the changes, when and why. The change set represents a logical change to a system and a means of creating any

version of a configuration that is not necessarily dependent on the latest version of that configuration [30]. The change set model supports a natural link to *change requests*. While change requests include information about a change, change sets contain the actual modifications making up a logical change. Although change sets are concerned with transactions, they do not support concurrency control. Thus, the change set model should be complemented with the check-out/check-in model [38].

Summary of SCM Models

This section has focussed on a description of four models that can be observed in commercial systems. From the four models outlined above, the check-out/check-in model and the composition model are useful for this research work. As the check-out/check-in model provides developers with the capabilities to efficiently maintain a repository through version control and concurrency control such as locking, version branching and merging, the model is appropriate for the version management of reusable components that exist within a repository. In addition, the composition model that supports system modelling and version selection, can be used for this research since it helps reusers and maintainers to compose a system using reusable components. On the contrary, the long transaction model focuses on the versions (configurations) of whole systems, therefore, the model might be useful for vendors that need to release many system versions or system variants for different platforms. The change set model that controls the evolution of a system using logical changes, supports management for the propagation of logical changes through a system family. Change sets provide the link to managing the change process through *change requests*. There is a need for a unified SCM model that provides a framework for all SCM models/concepts as a single SCM system may have difficulties meeting all needs throughout the software development/maintenance process.

2.2.5 Automation of SCM

Although SCM is sometimes performed manually and sometimes automatically, both manual and automated procedures can be useful for SCM. In the real world, these two procedures exist side by side. However, when manual procedures are heavily used, they can become burdensome and time-consuming. In general, whereas automatic SCM is good for large projects, manual SCM is effective for small projects and individual projects.

Babich [2] argues that "the greater the team size, the greater the possibility of error in manual procedures, and the greater the number of procedures that must be automated". Ben-Menachem [6] states that the techniques requiring automated SCM are *file locking, branches* and *merges.* These can only be implemented with the use of automatic SCM. He presents several problems with SCM that can be supported by an automated SCM system as follows: lack of information about changes; recurrence of bugs ('I fixed the bug already!'); confusions stemming from conflicting changes; unauthorised access and modifications; difficulties in regenerating old versions; incomplete or inaccurate system composition; and confusions resulting from system building which used incorrect versions.

There has been considerable progress concerning an automated SCM system in terms of environments (i.e., an SCM system) and tools (i.e., a stand-alone SCM tool). Many commercial packages for SCM have been announced and are currently available. In order to be able to make an SCM system effective and viable within a software development/maintenance environment, an automatic SCM system should be implemented as a unified model that can support most of the SCM concepts and models described in Section 2.2.4.

2.2.6 SCM Problems within a Maintenance Environment

SCM is a crucial solution for controlling the process of changes that occur during software development and maintenance. Whitgift [120] argues that "although the discipline of SCM is essential throughout the project life-cycle, it is never more important than during the maintenance phase". Software maintenance is concerned with changing an existing system and SCM provides precisely the framework that is needed to manage such changes. SCM problems are often most acute during the maintenance phase due to the existence of the largest number of software items to manage. Additionally, software items have many versions and many relationships exist between the items. For this reason, many of the functionalities described in the previous sections to illustrate aspects of SCM, need to be equally applied to the maintenance phase of a project.

Changes to items during the maintenance phase are more difficult than those during the development phase as a legacy system is probably maintained not by its developers but by someone else and even by new programmers. Thus, these changes must be carefully controlled in order to allow maintainers to work together efficiently and easily.

The version control facility of parallel development can be used for the maintenance phase as well as the development phase. It is common and good practice to divide a project team into two: a development team and a maintenance team. Once the completed system is first released to the end user, the maintenance team takes over responsibility for maintaining the system. The changes implemented by the maintenance team will be merged with other changes which will have been made in parallel by the development team [120]. Whereas the maintenance team usually deals with changes relating to *corrective maintenance*, the development team focusses on implementing changes associated with *perfective maintenance, adaptive maintenance and preventive maintenance* which are described in the following section 2.3.1. Without an automated SCM system, it is impossible to merge two

versions created through parallel development within a maintenance environment.

Software maintenance is considered the most expensive phase since most of the overall life-cycle cost is consumed in maintaining the system. Configuration control and version management amongst all SCM activities play a major role in initiating, evaluating, approving, and implementing change requests to a legacy system.

2.3   Software Maintenance

2.3.1   Introduction

It is clear that the key factor of a cost-effective information system (IS) is software maintenance because 60-80 percent of the total cost of the system life-cycle is spent during maintenance. *Software Maintenance* has been defined as follows:

> the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [60].

Until the 1980s, most efforts in the software industry had concentrated on software development whose objective is to produce a product that is on time and within budget while meeting user requirements, not necessarily a product that is reliable and maintainable. Software maintenance has not long been regarded as a creative activity and has not been chosen as a research topic by academia.

In recent years, maintenance has been investigated by the users of large systems in industry as well as academic researchers. The issue of software maintenance has to be dealt with, but it is usually more difficult than original development. Therefore, many organisations spend most of their time maintaining existing applications, i.e., legacy systems. The IEEE [59] categorised maintenance into four categories as follows:

- *Perfective maintenance* is the modification of a software product after delivery to enhance performance or meet new functionalities.

- *Adaptive maintenance* is the modification of a software product after delivery to enable a software system to adapt to a changed or changing environment.

- *Corrective maintenance* is the modification of a software product after delivery to fix discovered errors or faults.

- *Preventive maintenance* is the modification of a software product after delivery to prevent problems before they occur.

Munro [83] describes the distribution of the total maintenance activity together with four types of maintenance activity. His view of the types of maintenance is not different from the IEEE's definition. In addition, he states representations to be changed for each type of maintenance as follows: perfective maintenance requires changes to the requirements, design and code; adaptive maintenance needs changes to the design and code; corrective maintenance only requires changes to the code; and finally, preventive maintenance is associated with changes to the design and code. He presents the following distribution: perfective maintenance consumes 60% of the maintenance activity; adaptive maintenance consumes 18%; and, corrective maintenance and preventive maintenance consume 17% and 5%, respectively.

In Section 2.3.2, some software maintenance models are described. Section 2.3.3 shows a framework of a software maintenance support environment.

2.3.2 Software Maintenance Models

The traditional life-cycle model of a system has only considered the software maintenance activity as a single phase at the end of the cycle. It does not portray the evolutionary development and decomposed maintenance process that are very useful for most software systems [7]. In order to control software maintenance effectively,

it is necessary to divide the maintenance process into separate phases as in a development process. So far, many authors have proposed models for the maintenance process. The early models only give general guidelines whereas the latest ones define more detailed activities of the maintenance process. Boldyreff, Burd et al. [12] give a brief overview of some existing models such as Maintenance Assistance Capability for Software (MACS) [34], EPSOM [51, 37], the Durham Maintenance Model [55, 54], and REDO [115]. However, only EPSOM and the *Request-driven Model* are described in this section since these models are composed of well defined activities of a maintenance process and focus on establishing the procedures of change control.

EPSOM

EPSOM [51, 37] is a generic maintenance model. The model identifies 9 activities that form a maintenance-specific 'V' life-cycle. The 'V' model consists of change control (the left side: trigger, problem understanding, localisation, solution analysis, and impact analysis), change itself (the point of the 'V' model: implementation), and testing of the result (the right side: regression testing, acceptance testing, and re-insertion). Figure 2.1 shows the 'V' model that is revised slightly after being taken from the EPSOM model [51].

The steps of the process relating to change control are as follows [51]:

- Trigger—the maintenance process is initiated by a trigger, i.e. a program fault report or change request.

- Problem understanding—the maintainer has to determine whether or not the problem should be tackled.

- Localisation—the maintainer identifies precisely what has to be changed.

- Solution analysis—the solutions to implement change requests are devised.
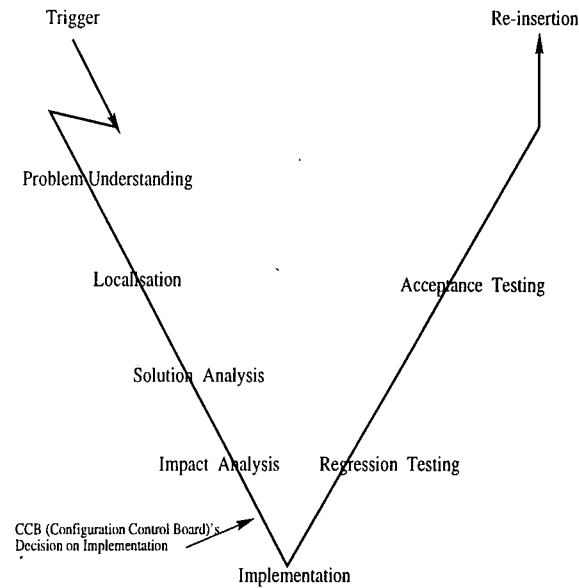
Figure 2.1: The EPSOM Model: a Maintenance-Specific 'V' Life-cycle

- Impact analysis—this aims at evaluating the consequences of the implementation of the changes in order to reduce unforseen side effects.

Request-driven Model

The Request-driven Model [7] identifies the activities of software maintenance initiated by user's change requests. The model has three major processes called *request control, change control, and release control.*

Request control deals with the user's change requests (CRs) in order to analyse, categorise and prioritise the CRs, and evaluate the impact of changes. The main activities associated with this process are as follows:

1. Collection of information about a change request.

2. Establishment of mechanisms to categorise the change request as either a corrective, adaptive, perfective or preventive type of change.

3. Use of impact analysis to evaluate the request with respect to cost/benefit.

4. Prioritisation of the change request.

Change control is usually considered the key process in this model as it includes the most expensive activity, i.e., the analysis of the existing applications. The activities of this process are as follows:

1. Selection of change requests from priority list.

2. Reproduction of the problem where appropriate.

3. Analysis of code, specifications and relevant documentation.

4. Design of the solutions for changes and construction of test cases.

5. Setting up of a quality control procedure, including V&V, review, inspection, etc.

Release control chooses the change requests that should be included in a new system version and makes the necessary changes to the source code. The activities relating to this process are:

1. Release determination.

2. Construction of a new release through editing source, archival and Software Configuration Management (SCM), and Software Quality Assurance (SQA).

3. Integration and regression testing.

4. Re-insertion (release).

5. Acceptance testing.

To conclude, the EPSOM model is a good framework for the maintenance process as it includes all maintenance activities associated with change control, change implementation, and testing of the result. The model forms a 'V' life-cycle and the format of the model is similar to a 'V' life-cycle for software development. For this reason, an organisation can easily introduce the model into a maintenance environment. Three processes of the request-driven model, in particular, request control and change control, are concerned with the configuration control activity which establishes the change control procedures to initiate, evaluate, approve and implement changes to an existing system. In addition, the Software Quality Assurance (SQA) activity can be performed through the configuration audit of SCM, i.e., Verification and Validation (V&V) as described in Section 2.2.3. Thus, the EPSOM model and request-driven model are most suitable for this research since they address the change control procedure well.

Bennett, Munro et al. [7] stress that an important part of the maintenance organisation is the Change Control Board (CCB) which reviews, prioritises, and approves change requests. As described in Section 2.2.3, changes to existing Software Configuration Items (SCIs) are made only via a formalised process controlled by the CCB. The CCB is the committee whose purpose is to control changes within a development/maintenance environment. Thus, the introduction of SCM into a maintenance environment or a maintenance model facilitates the successful management of the software maintenance process.

2.3.3   A Software Maintenance Environment

As shown in Figure 2.2, Kwon and Boldyreff [67] propose a software maintenance support environment that consists of several technologies and methods which can solve many problems within a legacy system. The software maintenance support
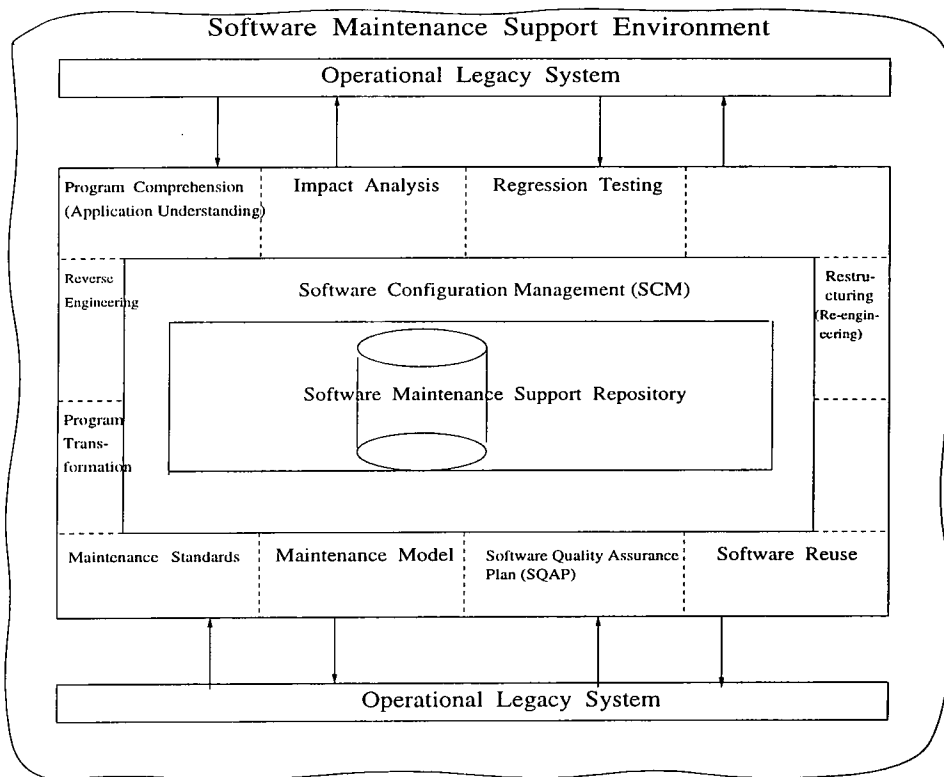
Figure 2.2: A Software Maintenance Support Environment

repository is a core part of the maintenance support environment and keeps tools associated with a variety of methods and technologies as well as program sources, specifications and documents reverse engineered using those tools. The repository is managed and controlled by a Software Configuration Management System (SCMS) whenever Software Configuration Items (SCIs) related to a legacy system are checked-out or checked-in.

A maintenance support environment has three major parts: the implementation part of maintenance that includes *Program Comprehension, Impact Analysis* and *Regression Testing*; the part of *Reverse Engineering* that can be supported by *Program Transformation* and *Restructuring*, and enables *Design Recovery*; and the part of the *Maintenance Model* and *Standards* that support the process and guidelines of software maintenance.

In order to make methods and tools successful, they need to be integrated into

a software maintenance environment as well as into a software development environment. In particular, methods and systems that support SCM, can be a good framework for constructing a maintenance environment.

2.4 Relationship between Reuse, SCM and Software Maintenance

There exist several similarities between reuse and SCM approaches to software engineering [68]. The common characteristics of the two approaches are as follows: the identification of a component-oriented approach; the use of a library; the construction of components; a close relationship with development and maintenance environments; interaction through Configuration Audit; relationship with standardisation; and both recognised as part of the SEI's Capability Maturity Model (CMM) [85].

A controllable basic unit of these two fields is an object or a component. Information on both can be stored in a repository, is subject to retrieval and requires change control. For instance, some information on version history and reuse history can be kept together in an identical component.

Although Whitgift [120] states that the software library is the core part of SCM as it contains everything relating to SCM such as source code, user and system documentation, test data, support software, specifications, project plans and derived objects, reuse also requires a software library which supports *'white box' reuse* and *'black box' reuse.* Changes to the library need to be controlled whenever they are made over time.

The construction of a new system using components must be an important issue for both reuse and SCM. A system modelling language such as a Module Interconnection Language (MIL) or a Component Description Language should be used to compose a system from software components.

Configuration Audit of SCM helps an organisation to evaluate software components since it ensures the completeness and correctness of a software product through the activities of Verification and Validation (V & V). 'Development/Design for Reuse (DfR)' also requires the process of evaluating produced software components using software quality metrics.

Standardisation is related to both SCM and reuse. In order to effectively create and use reusable components effectively, an organisation is required to standardise its development and maintenance processes including its reuse process. SCM helps an organisation to standardise and improve these processes. In order for a software process to produce a high quality product, it should be assessed and improved continuously. The process assessment and improvement programme can be successful only if the processes of software development and maintenance are standardised through the methodologies and principles of software engineering. As SCM is a good discipline to standardise the activities of the software development and maintenance processes, this research addresses how an SCM process can be introduced into an integrated model of the reuse process and maintenance process.

SCM is a minimum requirement for constructing a software development environment and maintenance environment as it provides a support function as well as a management function. Similarly, reuse is also concerned with a development environment/maintenance environment. The process of reuse needs to be integrated into an existing environment so that reuse can become viable, effective and systematic.

Finally, SCM is one of the key process areas at level 2 ('repeatable') of the CMM's 5 levels. Since a Reuse Capability Model (RCM) [32] is a guide to selecting improvement strategies by measuring current reuse capabilities and identifying the issues most critical to reuse improvement, the CMM software process maturity model can be applied to it. As most of the organisations are still placed on the 'initial' level of the CMM model, they require their processes to be moved towards a higher level in order to introduce 'systematic reuse' into a development/maintenance environment.

SCM is a central part of software maintenance because it is associated with changing the existing software and SCM is a discipline for controlling these changes. Zvegintsov [123] describes the role of SCM as "recording and creating linkages by which a maintainer can monitor and control the transition from change requests to the implementation and testing of the changes". He stresses the importance of SCM within a maintenance environment rather than a development environment.

Re-engineering and reverse engineering are both maintenance techniques and are vital to software reuse as they enable a maintainer to create/extract reusable assets from existing system components. In addition, a reuse process can support a maintenance process through 'Design/Development for Reuse (DfR)' and 'Design/Development with Reuse (DwR)'. There is a very strong similarity between software maintenance and reuse as these two processes require both SCM and program comprehension activities. Thus, the disciplines of maintenance and reuse are very cooperative and have characteristics in common.

As described above, because there exist similarities and relationships between reuse, SCM and software maintenance, we can solve many problems with reuse and software maintenance through an integrated reuse process and maintenance process within an SCM environment.

2.5   Conclusions

The objective of software engineering is to improve the productivity of software development and the quality of the systems produced. However, most of the total cost of the software life-cycle is spent on maintaining existing systems rather than developing new ones. If we want to improve software development, including maintenance, we should review the software maintenance process and establish an efficient procedure for a maintenance environment.

In this chapter, background research has been discussed with a view to introducing

SCM to the reuse library within a software maintenance environment in order to manage and control changes to both reusable components and existing systems, thereby achieving the high productivity of change implementation and producing a high quality product. Three fields related to this research, i.e., software reuse, SCM and software maintenance are very closely associated with one another as described in Section 3.3 of Chapter 3. The integrated approach can therefore be applied for these three disciplines of software engineering.

If multiple versions for each reusable component can be kept in a repository through the process of SCM, then the possibilities of reuse can be increased greatly. By reusing reusable component versions that have already been developed, an organisation can enhance its effectiveness in improving both the productivity and the quality of the produced software. However, it is clear that the true benefits obtained from reuse can only be achieved through *systematic reuse*, not *ad-hoc reuse*.

Although 'black box' reuse can be considered ideal reuse, this research also supports 'gray box' reuse that is an intermediate form of reuse, as well as 'black box' reuse and 'white box' reuse. Since a reuse library is subject to change over time a change control procedure must be established in order to manage all changes to reusable components within the reuse library.

Since both software reuse and maintenance as well as SCM require system modelling languages for software building, Module Interconnection Languages (MILs) and Component Description Languages have been reviewed and compared. PCL is considered the best configuration language in terms of several requirements. RE-SOLVE is the better Component Description Language as it can be used stand-alone and subsequently translated into implementation languages like C++ or Ada, and it enhanced Goguen's OBJ and LIL, and MILs developed thus far.

In order to implement systematic reuse successfully, an organisation should integrate the process of reuse into a software maintenance environment that consists of a number of technologies, methods and tools. Therefore, this chapter has also

reviewed most of the disciplines associated with a maintenance environment, and proposed a framework of a software maintenance support environment that includes a production library (repository) controlled and managed by SCM.

Since reuse should support a software development process or maintenance process, SEI's Capability Maturity Model can be applied to the Reuse Capability Model. An organisation can choose a suitable reuse method from the two methods (e.g. 'white box' reuse and 'black box' reuse) according to the level of reuse capability.

Commercial systems have introduced new SCM concepts into the classical principles of SCM (i.e., configuration identification, configuration control, configuration auditing and configuration status accounting). These new concepts are associated with three SCM models such as the composition model, the long transaction model, and the change set model, in addition to the initial check-out/check-in model. Of the four SCM models, three models (i.e., the check-out/check-in model, the composition model and the change set model) are suitable for the change control of reusable components within a software maintenance environment.

SCM functionalities are the foundation of any software development and maintenance environments. Good SCM capabilities help software reuse to be successfully introduced into an organisation because SCM expedites the integration of a reuse process into a maintenance/development environment. Configuration control and version control activities are related to this research since these activities provide reusers and maintainers with a means of controlling changes to reusable components and existing systems. Even though some SCM activities can be supported by manual procedures, most of the activities should be automated in order to reduce burdensome and time-consuming tasks.

This research will focus mainly on a software maintenance environment rather than a development environment. A software maintenance model needs to be combined and integrated with the model of the reuse process. The goal of this thesis is to develop an integrated model of the reuse process and maintenance process within

an SCM environment, and establish the change and version control procedures in order to manage changes to a reuse library and a legacy system.

In the next chapter, the rationale and motivation of this research are addressed in the order of 'new approaches to software engineering', 'problems with reuse and maintenance', 'similarities between reuse and SCM and between reuse and maintenance', 'direction of research', and 'originality of this work and discussion of similar work'.

Chapter 3

Rationale for an Integrated Model

The objective of this research is to develop an integrated model that can address many problems with software maintenance and software reuse through the functionalities of Software Configuration Management (SCM). There exist some similarities between software reuse and SCM. Reuse also has activities in common with software maintenance in terms of program understanding and SCM. This chapter discusses those similarities between these three fields that provide the rationale for this research.

In Section 3.1, new approaches to software engineering are summarised. Section 3.2 identifies some problems within a reuse library and a software maintenance environment that need to be solved to improve software productivity and software quality. Section 3.3 describes common activities and relationships between reuse and SCM and between reuse and maintenance. In Section 3.4, the directions of research that should be performed in order to tackle these problems, are described. Finally, Section 3.5 describes the necessity and value of this work, providing a discussion of other relevant work.

3.1 New Approaches to Software Engineering

Major changes in the way large-scale software-intensive systems are being developed, fielded and updated, have led to a new way of software engineering called *Component-Based Software Engineering (CBSE)*. The concept of designing and implementing software systems using a set of components has been proposed for at least three decades. Large-scale software development is increasingly achieved through the processes of component selection, evaluation and assembly. The components for system building are acquired from external suppliers. In order to pursue a component-based approach, many obstacles need to be overcome. Most of the problems stem from non-technical issues, but significant technical issues must also be tackled [17].

Basili [4] presented the fundamental concepts behind software process and product improvement using measurement and evaluation in an Experience Factory Organisation. He also successfully applied a concept of the *Experience Factory* to the Software Engineering Laboratory (SEL) at NASA/ Goddard Space Flight Centre. The Experience Factory aimed at capitalisation and reuse of experience and products from the software life-cycle. It is a logical and physical organisation whose activities are independent from the ones of the development organisation. In order to build an integrated environment for software development, the Experience Factory consolidates activities such as packaging experience, consulting, quality assurance, education and training, process and tool support, and measurement and evaluation.

A *software factory* is defined as "an environment which allows software manufacturing organisations to design, program, test, ship, install and maintain commercial software products in a unified manner" [80]. Software reuse is a crucial part of a software factory as the software factory is a factory system for the automatic production of software through a combination of software parts/components. There have been a wide variety of efforts to adapt the concept of the factory to software production by US companies such as GTE and IBM, and Japanese companies such

as NEC and Hitachi. By adapting existing reusable software components, the factory concept was primarily applied to producing large and well understood systems that were not conceptually new, such as computer operating systems and telecommunications systems [28, 23]. The software factory based on software reuse should be supported by a number of methods and techniques related to software engineering: *component engineering (i.e., reverse engineering and design for reuse), domain analysis, metrics, formal methods, project management, and software configuration management.*

A significant consideration is determining how these new approaches bring us better facilities for carrying out maintenance and reuse.

3.2   Problems with Reuse and Maintenance

Both Software Configuration Management (SCM) and reuse have long been advocated as means of achieving the enhancement of software quality and productivity. However, aside from a few success stories, these two approaches have not brought a significant result for the software engineering community, since there exist technical and managerial problems to be tackled in each and these approaches have been separately introduced into each organisation.

While reuse has problems associated with creating/recognising, cataloguing and retrieving reusable components as well as composing complex systems from those components and integrating a reuse process into any software development environment and maintenance environment, it also has some problems related to SCM such as how to control the change of reusable components, how to propagate the changes of a component to the reusers of the component, and how to control different versions.

As shown in Figure 3.1, a reuser and maintainer can use reusable components within a reusable software library through the three types of reuse: 'black box'
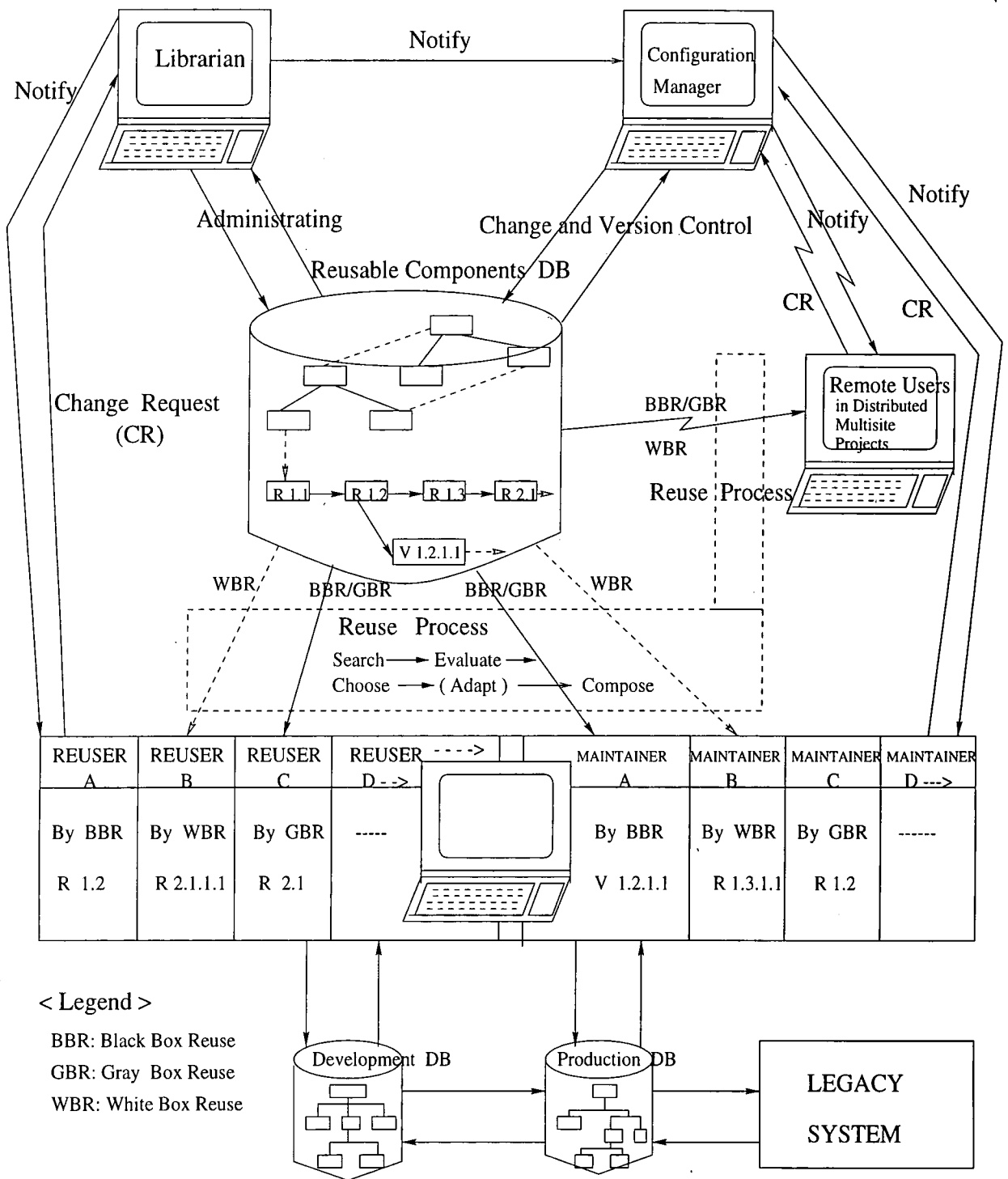
Figure 3.1: An Idealised Maintenance Environment

reuse, 'white box' reuse and 'gray box' reuse. A broken arrow in the figure shows the process of 'white box' reuse that requires the modification of reusable components before reuse, whereas a black arrow represents the process of 'black box' or 'gray box' reuse that uses reusable components on an "as-it-is" basis. A broken box shows the process of reuse supported by these three types of reuse. SCM practices are very useful for the control of 'white box' reuse but these can also be used for 'black box' reuse, as the repository that supports 'black box' reuse is subject to change over time. In other words, the reuse repository continues to evolve as a legacy system does, because the reuser and maintainer may ask a librarian or domain manager to change reusable components by using the change requests. There is clearly a need for *gray box* reuse between 'white box' reuse and 'black box' reuse, where it allows a few changes to reusable software components, e.g., renaming variables and changing parameters. Therefore, 'black box' reuse as well as both 'gray box' and 'white box' reuse require change control for changes to reusable components. After the reuser and maintainer successfully perform unit testing for modified reusable components, they need to check the components into the production library a tester or QA person uses to carry out integration testing before release.

If reusable components are used in several different projects and sites, the effect of changes to reusable assets will be much greater. As shown in Figure 3.1, whenever the changes to the reusable components have been made the librarian or domain manager should notify the reuser, maintainer and configuration manager as well as remote users that reusable components have been changed. The reuse library has to keep all the information on the change history and the reuse history acquired from the reuser and maintainer.

Most efforts in the software industry have concentrated on software development whose objective is to produce a product that is on time and within budget while meeting user requirements, not a product that is reliable and maintainable. Maintenance is usually more difficult than original development as changes to an existing software component may have many effects on other components and to carry out

70

an impact analysis is a difficult task. Thus, many organisations spend most of their time maintaining existing applications, i.e., legacy systems. A legacy system shown in Figure 3.1, has many problems with meeting user's requirements since the backlog of change requests (CRs) may be large. In this case, a reusable component database can be used to implement change requests (CRs) through 'black box' reuse and 'white box' reuse.

3.3 Similarities between Reuse and SCM and between Reuse and Maintenance

There exist several similarities between reuse and SCM approaches to software engineering [68]. As shown in Section 2.4 of Chapter 2, the common characteristics of the two approaches are as follows: the identification of a component oriented approach; the use of a library; the construction of components; close relationship with development and maintenance environments; interaction through Configuration Audit; relationship with standardisation; and both recognised as part of the SEI's Capability Maturity Model (CMM) [85].

There is a very strong affinity of software maintenance with software reuse. Major common activities relevant to both maintenance and reuse are SCM and system understanding. In other words, software maintenance and software reuse should be supported by two activities of SCM and system understanding. There exist common activities between the maintenance process and the reuse process such as "analyse CRs", "integration" and "re-insertion" that enable us to build an integrated model within an SCM environment. SCM enables a reuser and maintainer to solve some problems with reuse and maintenance [69]. In terms of SCM, software components within both an existing system and a reuse library are subject to change over time, so these components need to be controlled by the functionalities of SCM.

3.4   Direction of Research

The objective of this research is to construct an SCM procedure for a legacy system and a reusable software library which are actively maintained for use in a software maintenance environment. In other words, this research addresses an SCM process that supports an existing system through the reusable component repository that supports *'white box' reuse* and/or *'black box' reuse*, as well as SCM for the reuse repository itself. A 'black box' reusable component is reused without any modification, i.e., on an "as-it-is" basis, whereas in the case of 'white box' reuse, it may be modified before reuse.

This research presents the modelling work and prototype highlighting the importance of taking an extended view of configuration management where a reusable component library is used across a number of maintenance projects. Two kinds of changes need to be recognised. One is change to the products i.e. an existing system, and the other is change to the reusable components. The control and management of changes to the software components in a reuse repository, are critical to software product success. If the component is being used in multiple products, the effects of uncontrolled change are obviously more critical. The change control procedure for a legacy system using a reuse library is different from the traditional approach, so the maintainer should work in collaboration with the reuser who is very familiar with the reusable components within a reusable software library.

'Development for Reuse (DfR)' is the core part of the reuse process together with 'Development with Reuse (DwR)'; nevertheless, this research is associated with 'Development with Reuse (DwR)'. The objective of DwR is to catalog and retrieve reusable components, and develop a new system using them, whereas DfR aims at creating reusable software or re-engineering existing software to obtain reusable software. DwR can be supported by both 'black box' reuse and 'white box' reuse.

The main tasks of this research can be summarised as follows:

- Build an integrated process model of a reuse process and a maintenance process within a Software Configuration Management (SCM) Environment.

- Establish procedures of change control and version control for reusable components.

- Establish procedures of the reuse process for implementing change requests (CRs) to an existing system.

- Produce reports related to reuse and SCM such as a change history report, a reuse history report, a clear specification of a component, etc.

- Identify administrative functionalities associated with a reusable software library.

- Implement a prototype called TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets library) that supports the integrated process model named 'Maintenance with Reuse (MwR)'.

Via evolutionary development, one can make a rapid prototype and enhance it by redeveloping those portions where potential problems (functionality, performance, ease of use, maintainability, etc) are surfacing [15]. Therefore, this research will follow an evolutionary approach as a development method.

Chapter 4 presents the MwR model that supports three processes, i.e., the Configuration Management (CM) process, the reuse process, and the maintenance process. Chapter 5 describes TERRA's interaction with an SCM tool, CGI (Common Gateway Interface) and Web server, and tools used for implementation of the TERRA prototype. Chapter 6 shows the procedure for operation of TERRA by using several fill-out forms. Chapter 7 explores the MwR model by performing a scenario based case study. Chapter 8 describes an evaluation of the MwR model and tool using results and feedback obtained from the case study in terms of their benefits and limitations, and some amendments and customisation of the model and tool.

73

Finally, Chapter 9 presents general results and contribution of this research, and further work to be done.

Process Weaver [21] has been used for building the model since initial experiences with using ER (Entity Relationship) diagrams resulted in a complex model which was difficult to understand and adapt. Using Process Weaver, it is easier to draw and understand diagrams through the process of levelling down. Process Weaver is a tool for process modelling and process management, developed by Cap Gemini Innovation [21]. In Section 4.3, the model developed is presented and discussed.

3.5 The Originality of this Work and Discussion of Similar Work

From some problems and similarities described in Section 3.2 and 3.3, we need to solve some critical issues within reuse and maintenance, including population, retrieval, and change control of the reuse library and legacy system, through an integrated approach i.e., an introduction of SCM within the reuse process and maintenance process. This research develops an integrated model of the reuse process and the maintenance process within an SCM environment. The integrated model is called 'Maintenance with Reuse (MwR)' that supports maintaining a legacy system and an associated reuse library through functionalities of SCM.

Research into 'Development for Reuse (DfR)' and 'Development with Reuse (DwR)' have been carried out actively, but work on 'Maintenance with Reuse (MwR)' has never been done although maintenance is regarded as the most expensive phase of the software life-cycle. The difference between DwR and MwR is as follows: DwR is a development process supported by a reuse library whereas MwR is a maintenance process supported by a reuse library. Both DwR and MwR require a library that contains reusable components.

The MwR model consists of four major activities such as a configuration management (CM) process, reuse process, maintenance process, and administration of a reuse library. The activities of configuration management are a subsidiary functionality that enables one to integrate a reuse process with a maintenance process within a software maintenance environment. The CM process can also manage the evolution of a reuse library. Both processes of reuse and maintenance and the reuse library should be supported by SCM in order to manage changes to components which exist within these processes.

Several commercialised reusable software libraries and simple collections of reusable components have been announced, but most of them have not been successful as these libraries do not support further evolution or maintenance of the reusable components. In addition, most of the reuse libraries are difficult to use because they have not been linked to a development/maintenance environment. In recent years, several large reuse libraries that are usually sponsored by large organisations have been produced and are now available via the World Wide Web (WWW). The author has investigated four reuse servers such as AdaBasis, ELSA, ASSET and EUROWARE that have already been described in Section 2.1.4 and are now available on the WWW. As shown in Figure 3.2, the evaluation of these reuse libraries has been performed based on several important features such as *level of reuse, interoperability, retrieval method, change control, version control, usability, functionality, reports on change and reuse, certification of a reusable component, etc.* Although the table shows more details for evaluation of reusable software libraries, the functionalities of Software Configuration Management (SCM) are mainly discussed below for the purpose of this research work.

Interoperability which increases the availability of the asset pool, requires the same data model, classification schemes and terminology in order to share reusable assets between reuse servers [56]. Software Market, previously called ELSA, allows users to interoperate with other reuse servers such as ASSET, CARDS (Central Archive for Reusable Defence Software) and DSRS (Defence Software Repository

ITEMS	EUROWARE	ASSET	Software Market(ELSA)	AdaBasis
Level of Reuse	Software, Document	Software, Document	Software, Document, etc. (Metadata in MORE)	Software, Document
Flexibility	No	Good(Custom Digital Lib)	Good(Customer-Driven Lib Mechanism)	No
Interoperability	No	Partially Implemented	Partially Implemented	No
Way of Retrieval	Search by Classification, Free Text, Identifier	Search by Domain, Asset Name, Asset Identifier, Keyword	Search by Pattern and Natural Language	Domain
Change Control	a little	No	No	No
Version Control	No	No	No	No
Ease of Use	Easy	Easy	Easy	OK
Authorisation	On-line Registration Required	No Registration Required	No Registration Required	No Registration Required
Download	Free	Free/Charge	Free	Free
Reuse History Report	Yes	No	No	No
Evaluation of Components	No	Yes (4 Levels)	No	No
Number of Assets (As of Feb 1996)	52	Over 1,000	many(inc. ext. resources)	Over 174
Language Type of Code Assets	-	Ada, C++	Ada, C++	Ada based
Database	Flat File	Flat File	Multimedia DB(ie, MORE) Process Driven Repository	Flat File
Functionality(User Interface)	Moderate	Good	Good	Poor

Figure 3.2: Evaluation of Reusable Software Libraries

System). If ELSA metadata links are invoked, selections are transferred from the ELSA library to a temporary directory and made available to the reusers. This tri-lateral interoperation has not been fully implemented because of some limitations, but Software Market's MOREplus using SQLnet, has allowed the remote libraries to maintain and present their assets in the format they have in their libraries.

Only EUROWARE provides the reports on reusers' evaluation and history of reuse that might be valuable in encouraging reusers to use reusable components. Whenever reusers finish using reusable components successfully they need to record their views, feelings, findings and experiences about reusable components they used to meet their requirements.

The quality of a reusable component is crucial to the success of reuse. In order to improve the product quality and remove 'NIH (Not Invented Here)' syndrome from reusers, Science Applications International Corporation (SAIC)/Asset Source for Software Engineering Technology (ASSET) Company classified assets in 1 of 4 levels ranging from level 1 through 4 according to supplier's attestation, ASSET's review and confirmation, ASSET's testing, and ASSET's formal evaluation, respectively. Comparing with other reuse servers, ASSET can give more confidence to reusers who might have unwillingness in using reusable components.

Most reuse servers do not support completely the functions needed to control the evolution (i.e., revision and adaptation) of reusable components in a reuse reposi-tory. As an exceptional case, EUROWARE (Enabling Users to Reuse Over Wide Areas) provides a reuser/user with the function of a change request (CR), but does not support the process for implementation of the CR. Thus, the CR has no corre-sponding version number although each reusable component keeps only one version number. In terms of change and version control, reusable components of ELSA keep one version number, but ELSA does not support any other change control. In particular, there are no reuse servers which support efficient version control within a maintenance environment. All reuse servers described above have nothing to do with the software development and maintenance environments [70].

The reason why this research considers reuse and SCM within a software maintenance environment, not a development environment, is that 60-80 % of the software life-cycle cost is spent on a software maintenance phase and reuse can potentially help to reduce the cost here. In addition, SCM is a crucial part of software maintenance since it is associated with changing existing software components. In particular, SCM is a discipline for controlling these changes.

The rationale of this research is as follows: given the high degree of commonality between reuse and maintenance and between reuse and SCM, and given many of the technical problems associated with reuse in a maintenance environment can be solved by applying the discipline of SCM, we have concluded that the processes of component reuse, component maintenance and reuse library management all need to be integrated within an SCM environment to achieve effective long term support for all of these processes [69].

3.6 Summary

To summarise, this chapter has described new approaches to software engineering, discussed the problems with reuse and maintenance that need to be solved before reuse can effectively be introduced into a maintenance environment, and identified the similarities between reuse and SCM and between reuse and maintenance that enable an integrated approach of reuse and maintenance within an SCM environment. In addition, the chapter has shown the directions of this research, including the objective, main tasks and method of this research work. Finally, the innovative aspects of the MwR model to be developed have been discussed, in comparison with other reuse servers currently available on the Internet. The next chapter will address the modelling aspects of this research.

Chapter 4

Modelling of Maintenance with Reuse (MwR)

Process modelling is critical to the software process assessment and improvement cycle as it is composed of activities which produce a software product. In this chapter, a process model of an integrated approach has been created using Process Weaver. The chapter describes a process model to support the maintenance environment with a reuse activity, applying Software Configuration Management (SCM) disciplines to a library of reusable software components. The MwR model consists of three major processes—a reuse process, a maintenance process and an SCM process. Thus, the MwR model has been produced and refined by an automated tool for process modelling, Process Weaver that enables both consistency to be maintained between processes and the complexity of models to be reduced.

Section 4.1 outlines the definitions of concepts related to process modelling and describes the necessity of an automated tool for process modelling. In Section 4.2, the structure and notations of Process Weaver that has been used for modelling of this research, are summarised. Finally, Section 4.3 describes the detailed activities of the MwR (Maintenance with Reuse) model that consists of a Configuration Management process, a reuse process, a maintenance process, and administration of a

reuse library. This section also identifies the relationships between the reuse process and maintenance process, and presents the concepts associated with a product line.

4.1 Overview of Process Modelling

Software process modelling is a branch of software engineering whose aim is to improve the process of software development in order to enhance the productivity and quality of a software product, and to reduce the costs of software development. Dowson and Wileden [35] state that *software process* is a collection of related activities that are seen as a coherent process and involved in the production of a software system. They also define *software process model* as a purely descriptive representation of the software process. This model should represent attributes of a range of particular software processes.

Snowdon [99] defines *Process Modelling* as "the production of models of software development/maintenance processes and the use of these models in an Integrated Project Support Environment (IPSE)". The process modelling concepts are as follows:

- The *production process* is the set of "external" production elements such as real world activities, artifacts, tools, agents, roles, and embedding projects. It is also part of the process for developing and maintaining the product to be delivered to users.

- The *meta-process* is the set of "external" meta-elements that maintain and evolve the whole process, i.e., the production process, the meta-process, and the process support.

- The *process support* is the "internal" process model and the technology to define, modify, analyse, and execute it. The latter contains process modelling methods and languages, process modelling tools and process model in-

terpreters.

- The *process model* is "a description of a process expressed in a suitable process modelling language. A model is always an abstraction or a partial description of the reality that the model represents.

- The *process modelling language (PML)* is a formal notation used to describe process models, both for the production process and meta-process.

The production process and the meta-process are entities of the external real-world whereas the process support produces internal computer models to govern both the production process and the meta-process [99]. The created process models should be manipulated by a process tool that is exemplified in Section 4.2 in order to control and support the evolution of the real world. Manual implementation of process models can be error prone and time consuming. Christie [23] argues that software process automation by a process tool has made dramatic impacts in improving software productivity and there is no reason why similar approaches cannot be adapted to a process of software production. Software process automation has only recently become practical as a result of the widespread use of personal computers and workstations, and the growth of networking capability, which can lead to powerful distributed computing and human communications. In this research, Process Weaver [21] has been used to identify requirements for the processes, define process models, and verify process models.

4.2   Introduction to Process Weaver

In this research, the process model has been developed using Process Weaver [21]. Process Weaver[1] consists of a Work-context Editor, a Cooperative Procedure (i.e.,

---

[1]The author would like to thank Cap Gemini Innovation for allowing me to use Process Weaver in this research.

processes) Editor, a Method (i.e., activity hierarchies) Editor, and an Activity Editor. The Work-context Editor is used to model the working environment of a user; what to do and how to do it. The Cooperative Procedure Editor models the process part of a terminal or a refined activity in terms of *sequence, synchronisation and parallelism*. The Method Editor is used to describe the way software development has to be performed. It breaks methods into activities. Using the Activity Editor, one can specify task inputs, outputs and roles for these activities. Since the Work-context Editor is usually related to communication between team members and identifying specific roles relevant to carrying out the work, only the outputs of the Cooperative Procedure Editor and the Method Editor are described in the following section.

In order to better understand diagrams produced using the Cooperative Procedure Editor, 6 conditions and 7 actions that are used for editing of the cooperative procedure model, are described below. The number of the action or condition type is mainly matched to that of the action or condition shown in Figure 4.14 of Section 4.3.4.

The cooperative procedure editing window of Process Weaver displays several icons composing the palette on the left hand side of the working area. The palette is decomposed into three regions grouped as Transitions (i.e., Conditions, Actions) and Places. A cooperative procedure model that uses the Petri-net based formalism, is represented as a set of places and transitions with at least one initial place. An initial place is marked when the cooperative procedure starts. A transition is composed of a condition part and an action part. The condition part is used to enable the execution of the action part only if the condition is met [21].

The editor provides six types of condition as follows:

1. **Work-context condition** is used to specify that the cooperative procedure is waiting for an answer from a work-context previously sent to a user. The condition is shown in the upper part of the transition numbered 1 in Figure 4.14.

2. **Procedure condition** is used to indicate that the cooperative procedure is waiting for a given state reached by another cooperative procedure previously launched. The condition is shown in the upper part of the transition numbered 2 in Figure 4.14.

3. **Event condition** is used to describe the synchronisation between the cooperative procedure and the external world. The condition is waiting for an external event. The condition is shown in the upper part of the transition numbered 3 in Figure 4.14.

4. **CoShell condition** is used to express a logical expression. The condition is shown in the upper part of the transition numbered 4 in Figure 4.14.

5. **Collect condition** is used to show how to collect answers from a group of users who previously received work-contexts. The condition is shown in the upper part of the transition numbered 5 in Figure 4.14.

6. **Empty** is used when the transition has no condition. The condition is shown in the upper part of the transition numbered 6 in Figure 4.14.

The editor provides seven types of action as follows:

1. **Empty** is used when the transition does not perform any action. This action is shown in the lower part of the transition numbered 1 in Figure 4.14.

2. **Procedure action** is used to launch asynchronously a sub-procedure, which becomes a child of a specific cooperative procedure at run-time. This action is shown in the lower part of the transition numbered 2 in Figure 4.14.

3. **Synchronous Procedure action** is used to start synchronously a sub-procedure. When this action is taken the execution of the current procedure is temporarily halted until the sub-procedure terminates. This action is shown in the lower part of the transition numbered 3 in Figure 4.14.

4. **Event action** is used to send an event. This action is not shown in Figure 4.14.

5. **Distributing action** is used to send a work-context to a group of people. This action is shown in the lower part of the transition numbered 6 in Figure 4.14.

6. **Work-context action** is used to send a work-context to a person. This action is shown in the lower part of the transition numbered 7 in Figure 4.14.

7. **CoShell action** is used to express specific functions using the CoShell (Cooperative Shell) language. This action is shown in the lower part of the transition numbered 8 in Figure 4.14.

Process Weaver enables automation of the process of modelling that is difficult and error prone. Every activity of models created using Process Weaver's Method Editor is automatically linked to its procedure model developed using the Cooperative Procedure Editor so that the models produced by the two editors (i.e., the Method Editor and the Cooperative Procedure Editor) can be kept consistent with each other.

4.3 Process Model of Maintenance with Reuse (MwR)

Figure 4.1 shows the process model for *'Maintenance with Reuse (MwR)'* decomposed into activities using the Method Editor of Process Weaver. 'Maintenance with Reuse (MwR)' is used instead of 'Development with Reuse (DwR)' since the activities of software reuse have been integrated with the activities of SCM in order to support the process of software maintenance.

MwR includes four major activities: *configuration management, reuse process, maintenance process and administration of a reuse library* [69]. The reuse and maintenance processes are associated with controlling evolution of legacy systems
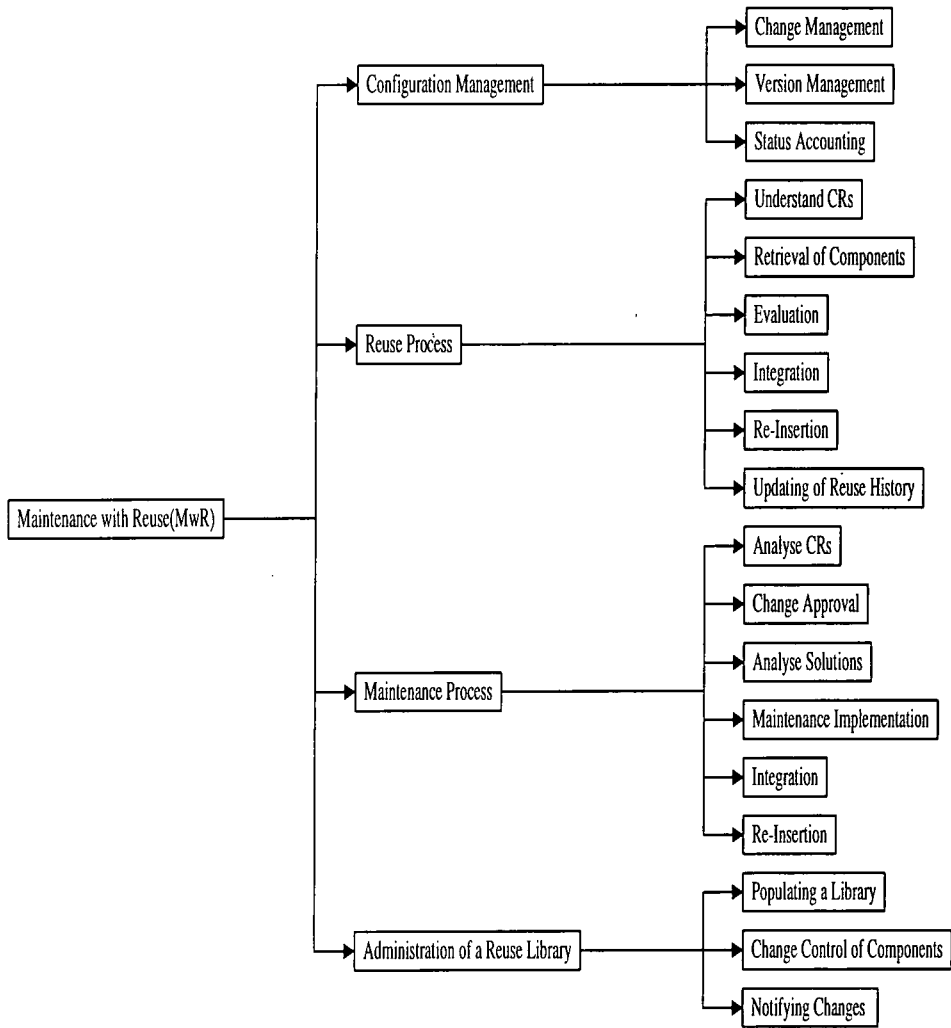
Figure 4.1: The Activity Decomposition of Maintenance with Reuse (MwR)

whereas the process for administration of a reuse library is concerned with managing population and evolution of the reuse library. The process of configuration management supports the other three processes as their subsidiary functionality. Figure 4.2 provides an alternative view of the activity decomposition in order to identify relationships between the four processes. In general, configuration management is performed by a configuration manager, the activities of a reuse process by a reuser, the activities of a maintenance process by a maintainer and finally, the activities of administration by a librarian, or domain manager.

The librarian/domain manager is responsible for the acquisition of new reusable components, and the definition, evaluation, classification, population, evolution and deletion of reusable components. Dabrowski et al. [29] define a *domain manager* as "an individual or organisation responsible for managing the definition, use, evaluation, and evolution of assets within the domain". However, since the librarian can take over the tasks of the domain manager in the reuse process, the author has used the terms librarian and domain manager interchangeably. The configuration manager has responsibilities for approval of change requests, change of components, propagation of changes and maintaining of version history for a legacy system.

The role of a reuser within a maintenance environment is to reuse reusable components to maintain a legacy system, to issue change requests to reusable components, to request new components, and to update reuse history. A maintainer is responsible for maintaining an existing system and issuing change requests to existing components. As this research focuses on SCM for a *Reusable Software Library (RSL)* that supports a software maintenance environment, the reuser in the process model could take over the role of a maintainer. However, the organisation of software development could have both a reuser who works on developing a software system using reusable components and a maintainer who implements change requests to an existing system using reusable components. In addition, when the reuse technology is first introduced into the organisation, both a reuser and a maintainer are necessary in order to standardise the activities of the reuse process and to encourage a main-
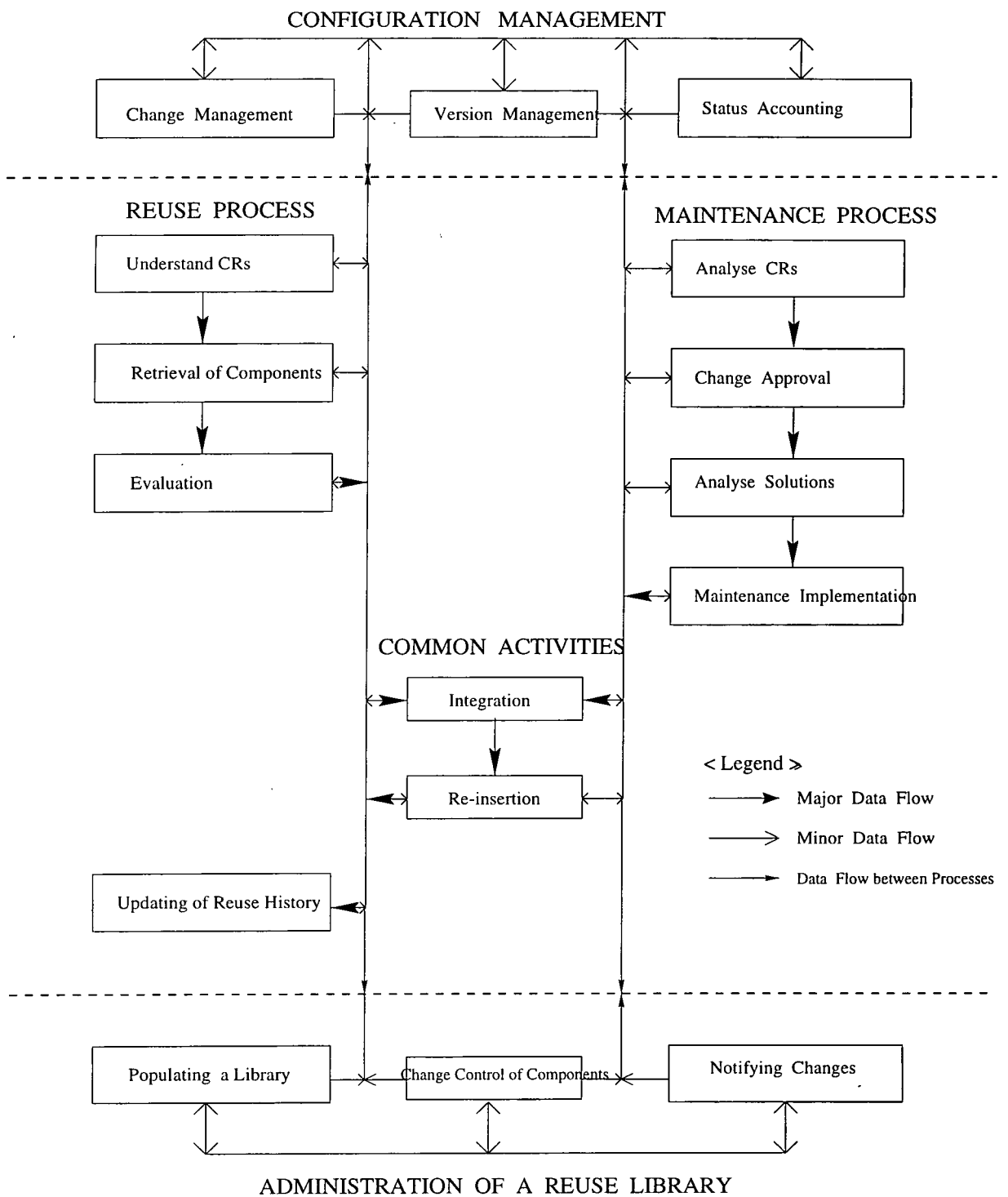
Figure 4.2: The Activity Decomposition of Maintenance with Reuse (MwR)

tainer who is not willing to reuse, to reuse reusable components for maintenance of an existing system. Each terminal activity (e.g., version management) that composes a refined activity (e.g., configuration management) is discussed below in more detail.

4.3.1 Configuration Management

As shown in Figure 4.1, 'configuration management' consists of activities such as '*change management, version management and status accounting*'.

'Change management' establishes the change control procedure to initiate, evaluate, approve and implement changes to a baseline. In particular, this activity is responsible for approving a change request that can be issued by a reuser, a maintainer, and an end user.

Software components are subject to evolution after release as they need to be corrected, adapted and enhanced, so both an existing system and a reuse library require 'version management' for their components. These changes create new revisions and these revisions are time-ordered. A variant is used to meet similar but different requirements at the same time. For example, there might be two variants of a module in order to fulfil slightly different requirements for two kinds of platforms. Whitgift [120] classified variations into two categories: *temporary variants* that are eventually merged into the main line and *permanent variants* that are never merged.

The objective of 'status accounting' is to provide a maintainer/reuser with visibility by recording and reporting the status of all components and change requests. Thus, in this research the activity of status accounting provides most information relevant to reuse and SCM for a project manager, a configuration manager, a maintainer, a reuser, and a librarian/domain manager. Each reusable component should include some information on specification, quality and administration. A change history should be recorded, such as who made the changes, what changes have been

88

made, when the changes were made, and why the changes were made. Although different people need different SCM information at different times and in different forms, Ben-Menachem [6] presents a minimal wish list of reports as follows: transaction log, change log, item 'delta' report, resource usage, stock status (i.e., status of items), changes in progress, and deviations agreed upon.

4.3.2   Reuse Process

The objective of this section is to establish procedures of the reuse process for implementing change requests (CRs) initiated from an existing system. Figure 4.1 shows that 'a process of reuse' is composed of *understand CRs, retrieve components, evaluate, integrate, re-insert, update reuse history.*

As shown in Figure 4.3, 'the understanding of the CR' includes two processes: 'identifying the requirement of the CR' and 'determining which software components should be retrieved' to meet the requirement. If a domain analysis has already been carried out the process of understanding can be easily performed. The domain analysis is described in more detail in Section 4.3.5. For instance, the name of the domain can be used effectively as a keyword for search.

'The retrieval of components' consists of 'deciding on a search method' (i.e., keywords, classification, etc.), 'retrieving components' and 'retrieving component versions' as shown in Figure 4.4. A search mechanism supporting queries enables reusers to find a set of components which match the identified requirements. Ideally with a sufficiently well populated library and appropriate search and retrieval mechanisms in place, a large number of relevant components will be found. After retrieving reusable components, a reuser also extracts reusable component versions by using the history of a change. The versions of these components can then be reviewed for their suitability.

Using the reuse history and statistics, the reuser evaluates alternative versions

89
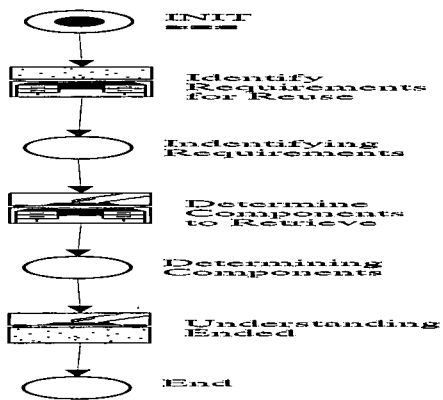
Figure 4.3: The Process for Understanding of the CR
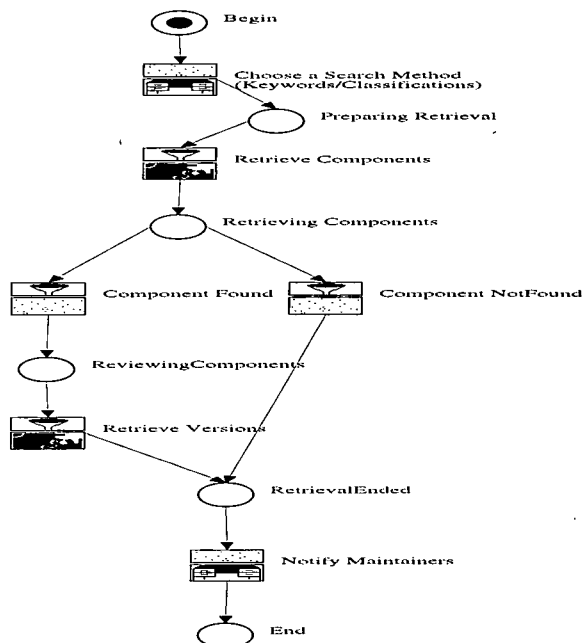
Figure 4.4: The Process for Retrieval of Components

90
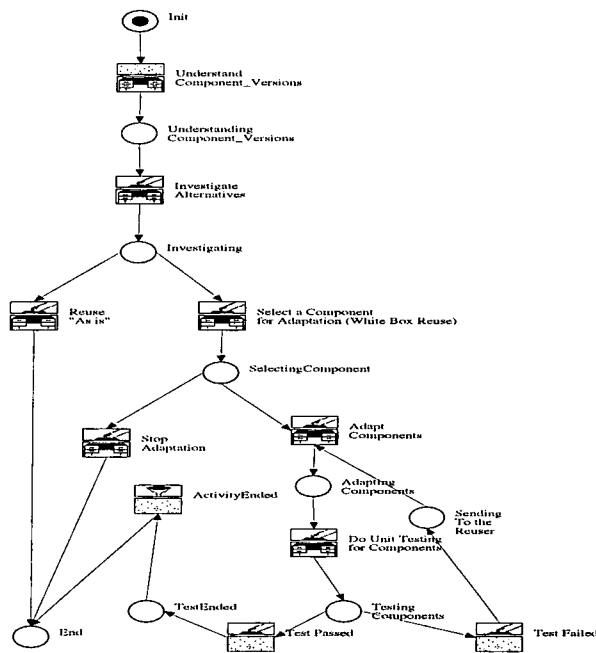
Figure 4.5: The Process for Evaluation of Components

of each reusable component and chooses the best version for his needs. Figure 4.5 shows that 'the evaluation of components' includes four sub-activities: understanding retrieved component versions, investigating alternatives to fit requirements of reuse, selecting a suitable component version for reuse and possibly adaptation, and adapting a chosen component where appropriate. After comparing reusable component versions, a reuser has to adopt either 'white box' reuse or 'black box' reuse depending on the level of adaptation/modification of retrieved reusable component versions. The reuser may modify reusable component versions if necessary for 'white box' reuse. Only if the estimated effort to modify a reusable component from the repository does not surpass the effort to adapt an existing legacy component, the reuser will adopt 'white box' reuse. The process of adaptation is similar to the process of maintenance as it includes the processes of modification and testing. After modifying a reusable component, the reuser performs unit testing of the component.

As shown in Figure 4.6, 'the integration of components' consists of sub-activities

91
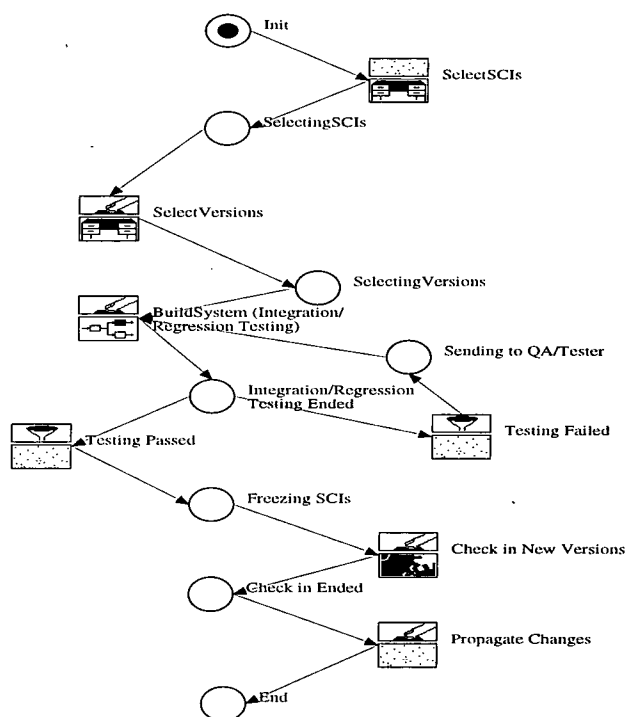
Figure 4.6: The Process for Integration of Components

such as selecting reusable components, selecting component versions, performing integration testing and regression testing, freezing the new baseline, and propagating changes. The objective of integration testing is to validate chosen and modified reusable components in order to ensure that the reusable components work correctly with an application. The reuser integrates the chosen component versions to build a system/subsystem using composition rules. It is desirable for the reuse library to support the function of system building that constructs a system using reusable components. In general, the process of system building consists of a system model and configuration thread (version selection rules). A system model lists all the components that make up a system. The system model describes the relationships between the system's source and derived items as well as between the components of a system. A configuration thread specifies which revision should be chosen for each component to compose a specific system configuration. A Syntactic Interconnection Language such as SySL (System Structure Language) and PCL (Proteus Config-
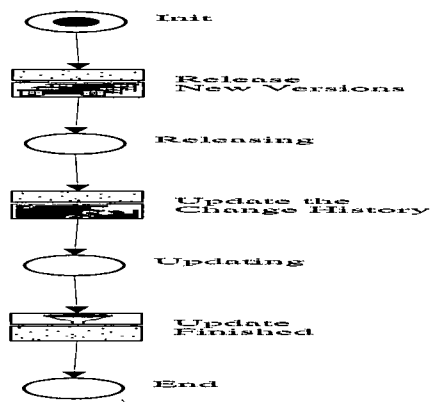
Figure 4.7: The Process for Re-insertion of Components

uration Language) may be used to construct composition rules for an integration test. The objective of regression testing is to verify that the modification made by 'white box' reuse has not caused any side effects. The integration testing followed by 'black box' reuse does not require any regression testing as 'black box' reuse does not entail any modification. The regression testing should also be performed using the results from an impact analysis.

Figure 4.7 shows that 're-insertion' includes 'releasing new versions' and 'updating the change history'. In general, the change history is automatically recorded by an SCM tool. The process of 're-insertion' is exactly the same as that of a maintenance process as will be shown in Section 4.3.3.

If the release of a new system is performed successfully the reuser updates files related to reuse history. As shown in Figure 4.8, 'the updating of reuse history' consists of 'updating reuser's evaluation file', 'updating a reuse experience file' and 'updating a statistical file'. 'The updating of reuse history' facilitates further reuse of the reusable components. In addition, it enhances the reusability of the component because the accumulated experiences of reusing the component can increase users' confidence in the components.

93
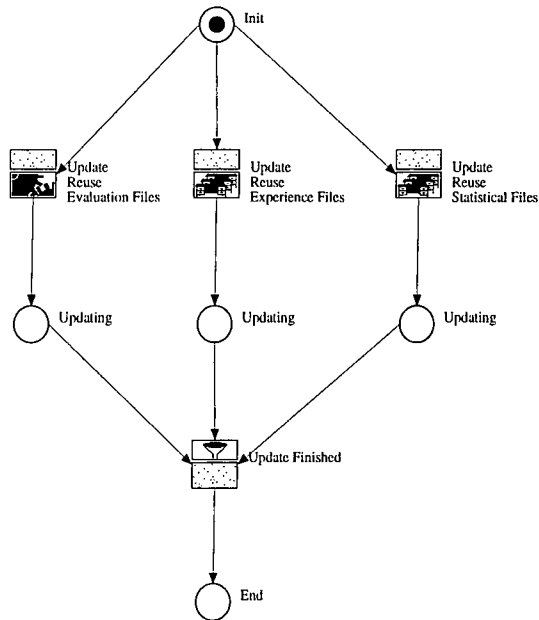
Figure 4.8: The Process for Updating of Reuse History

Figure 4.9 describes a combined reuse process that contains all the reuse activities and includes the concept of the component version. The process is similar to the reuse process described above except that it has separated 'retrieval of reusable component versions' and 'adaptation of reusable component versions' from 'retrieval of reusable components' and 'evaluation of retrieved components versions', respectively. The process does not include the process of 're-insertion'.

4.3.3 Maintenance Process

As shown in Figure 4.1, 'the maintenance process' consists of *analyse CRs, change approval, analyse solutions, maintenance implementation, integration and re-insertion.*

Since this research focusses on a software maintenance process, it is assumed that the change request is produced during maintenance, not development. Therefore, a change request can be issued by a maintainer or an end-user. As shown in Fig-
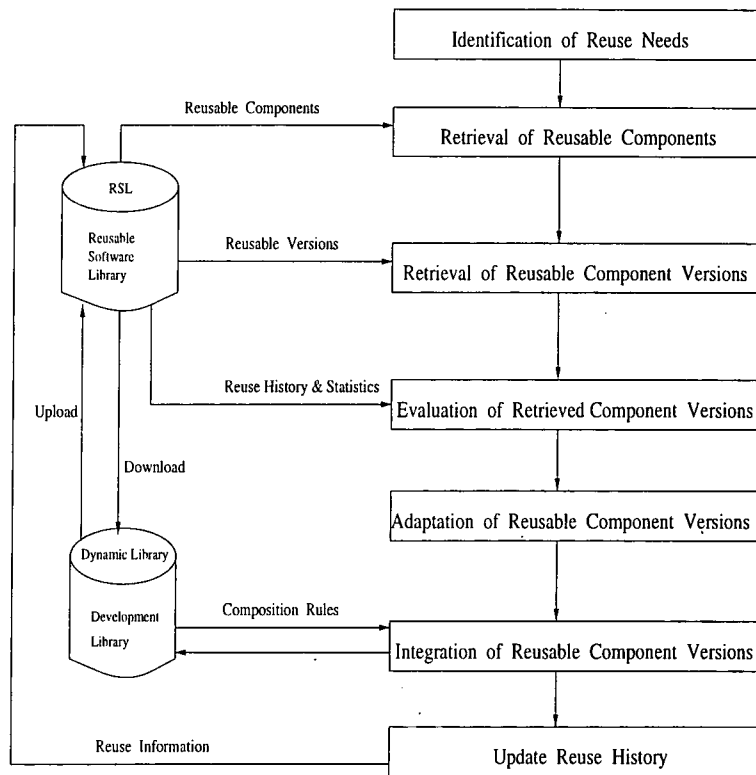
Figure 4.9: The Reuse Process Incorporating Version Control

ure 4.10, the process of 'analyse CRs' consists of 'receive CRs, analyse CRs, write an Engineering Change Proposal (ECP), and submit an ECP to the CCB'. Every change request is forwarded to a maintainer, a reuser, and a configuration manager at the same time. After the maintenance team receives the CR it should analyse the CR in order to decide whether the CR resulted from incorrect operation or misunderstanding of the system.

Figure 4.11 shows that the process of 'change approval' includes 'receive ECPs, analyse ECPs and approve ECPs'. This process is performed by a CCB (Configuration/Change Control Board) whose purpose is the control of changes. To make an efficient decision, the CCB needs to have authority and expertise. Thus, it should be composed of a configuration manager, a project manager, a QA manager and representatives from both a maintainer and an end-user.

As shown in Figure 4.12, the process of 'analyse solutions' consists of 'identify
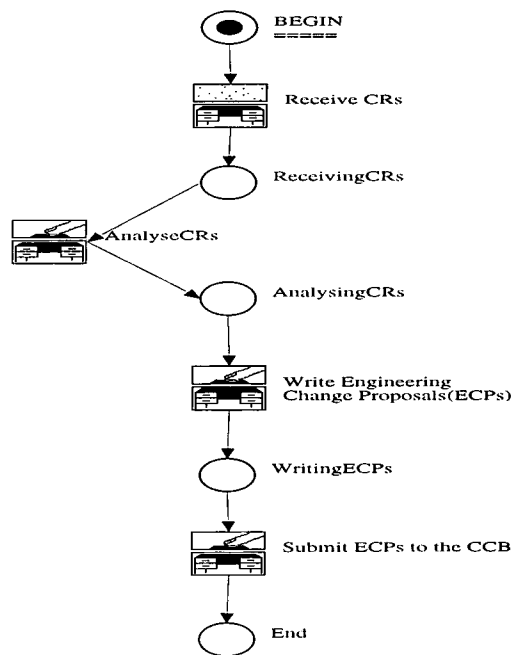
Figure 4.10: The Process for Analysis of CRs

solutions, perform an impact analysis, estimate resources, and plan implementation of changes'. If a maintainer can have several solutions to the CR he needs to estimate and compare the cost of implementing each solution. The results from the impact analysis can be used to estimate manpower and cost.

As shown in Figure 4.13, the process of 'maintenance implementation' includes 'request components, allow check-out, check-out components, modify components, and perform unit testing'. The subprocess of 'modify components' is similar to the subprocess of 'adapt components' within 'evaluation of components' of the 'reuse process'. Whereas the CCB is responsible for monitoring the implementation of the change requests (CRs) if they are approved, a project manager is in charge of implementing approved CRs by the due date using the resources allocated by the CCB.

The activity of 'integration' of the maintenance process is exactly the same as
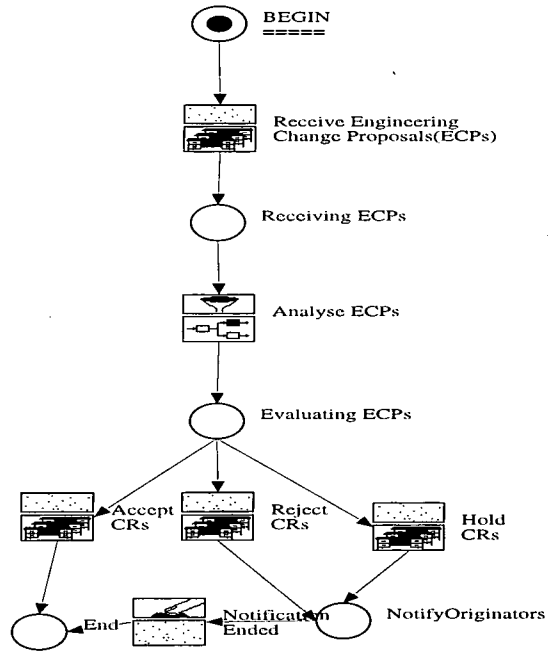
Figure 4.11: The Process for Approval of Changes
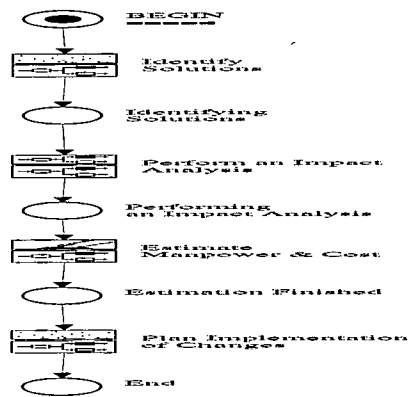
Figure 4.12: The Process for Analysis of Solutions

97
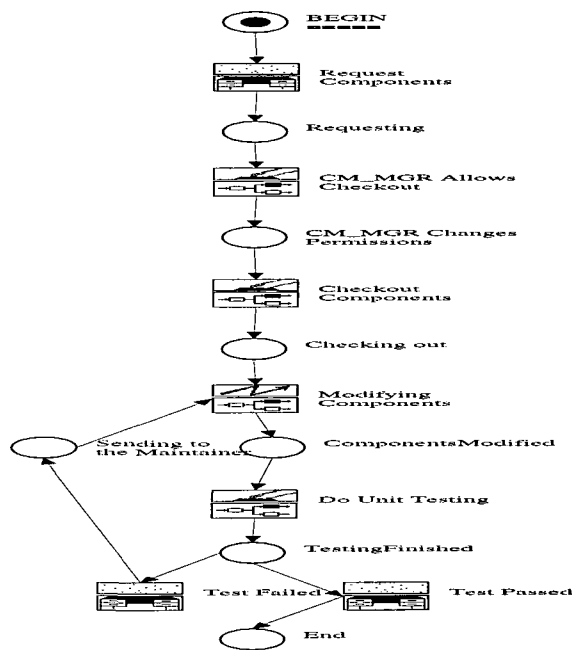
Figure 4.13: The Process for Implementation of Maintenance

that of 'integration' of the reuse process except that the 'regression testing' of the
maintenance process is always required although the reuse process only needs the
'regression testing' when 'white box' reuse is applied to the process. Figure 4.6
shows that the process of 'integration' consists of 'selecting reusable components,
selecting component versions, performing integration testing and regression testing,
freezing the new baseline, and propagating changes'. If the above two tests are
completed successfully, the new baseline is established and a configuration manager
notifies a project manager, a librarian/domain manager and maintainer that the
new configuration version is available.

The process of 're-insertion' consists of 'release new versions and update the
change history'. The activities of 're-insertion' are exactly the same as those of
're-insertion' within the reuse process as shown in Figure 4.7.

4.3.4 The Relationships between the Reuse Process and the Maintenance Process

Figure 4.14 shows the cooperative procedures of the reuse process and maintenance process developed from Figure 4.2. In order to show the integrated process of the reuse process with the maintenance process, three processes of the MwR model, i.e., 'configuration management, reuse process, and maintenance process' have been combined with one diagram that is similar to an extended Data Flow Diagram (DFD). This procedure assumes that a configuration manager together with a maintainer is involved in activities for evaluating and approving of change requests. As this research is based on a software maintenance environment, most activities of the reuse process have relationships with those of the maintenance process. For example, the analysis of the CR within a maintenance environment is associated with subactivities of both the maintenance process and reuse process. Therefore, change requests are passed on to both a reuser and a maintainer for reviewing at the same time.

The maintainer shown in Figure 4.14 knows all information about an existing system whereas the reuser shown on the left side of the figure is an expert on the components in a reuse library. The reuser works in collaboration with a domain manager who is familiar with the components within a domain and who can provide detailed information to the reuser.

As shown on the left side of Figure 4.14, after the reuser receives a change request he can start with identifying reuse requirements. Then the reuser searches for reusable components that match the reuse requirements and also retrieves component versions using the version history. Although the reuser may find reusable component versions from the library, he should await the answer from the maintainer, where the change proposal is approved, disapproved, or held by the CCB. If reusable component versions are found from the repository and the CCB approves the change proposal, then the next step, the evaluation of component versions will be performed, followed by activities of integration, re-insertion and recording of reuse

history. If reusable component versions do not exist in a repository, then the reuser notifies the maintainer of the fact and the process of reuse is ended. In other words, if there is no reusable component that can be reused or modified for 'black box' or 'white box' reuse, then next processes, i.e., activities of modification, integration and re-insertion must be performed by the maintenance process and the maintainer should take over subsequent processes.

The maintainer and configuration manager analyse a change request and decide if it comes from an incorrect operation, incorrect documentation, or incorrect software. Then, they produce an Engineering Change Proposal (ECP) to be submitted to the CCB (Configuration Control Board). ECPs can be classified as "emergency", "urgent", and "routine". The maintainer and configuration manager deal with change requests in close cooperation with a reuser. The configuration manager is responsible for monitoring of processing and dealing with all change requests. The CCB is responsible for approving, disapproving/rejecting, or holding a submitted change proposal.

Although a change proposal is accepted by the CCB, a maintainer should wait until a reuser finds out whether reusable component versions related to the change request exist or not. If the change proposal is rejected, the CCB should notify an originator and a reuser with an appropriate reason. If the change proposal is pending, the CCB should analyse it again at a later time and also notify a reuser. If the CCB accepts the change proposal and reusable component versions do not exist in the repository, the next activities of the maintenance process continue. If the CCB accepts it and reusable component versions exist, the reuser continues to perform the remainder of the activities of the reuse process, while the maintainer stops carrying out the process of maintenance.

Where reuse is not possible, i.e, no components have been retrieved, and the CCB has accepted the change proposal, then the maintainer analyses a report on change approval in more detail, develops a specific solution for approved requirements, and also identifies Software Configuration Items (SCIs) that are affected/impacted by
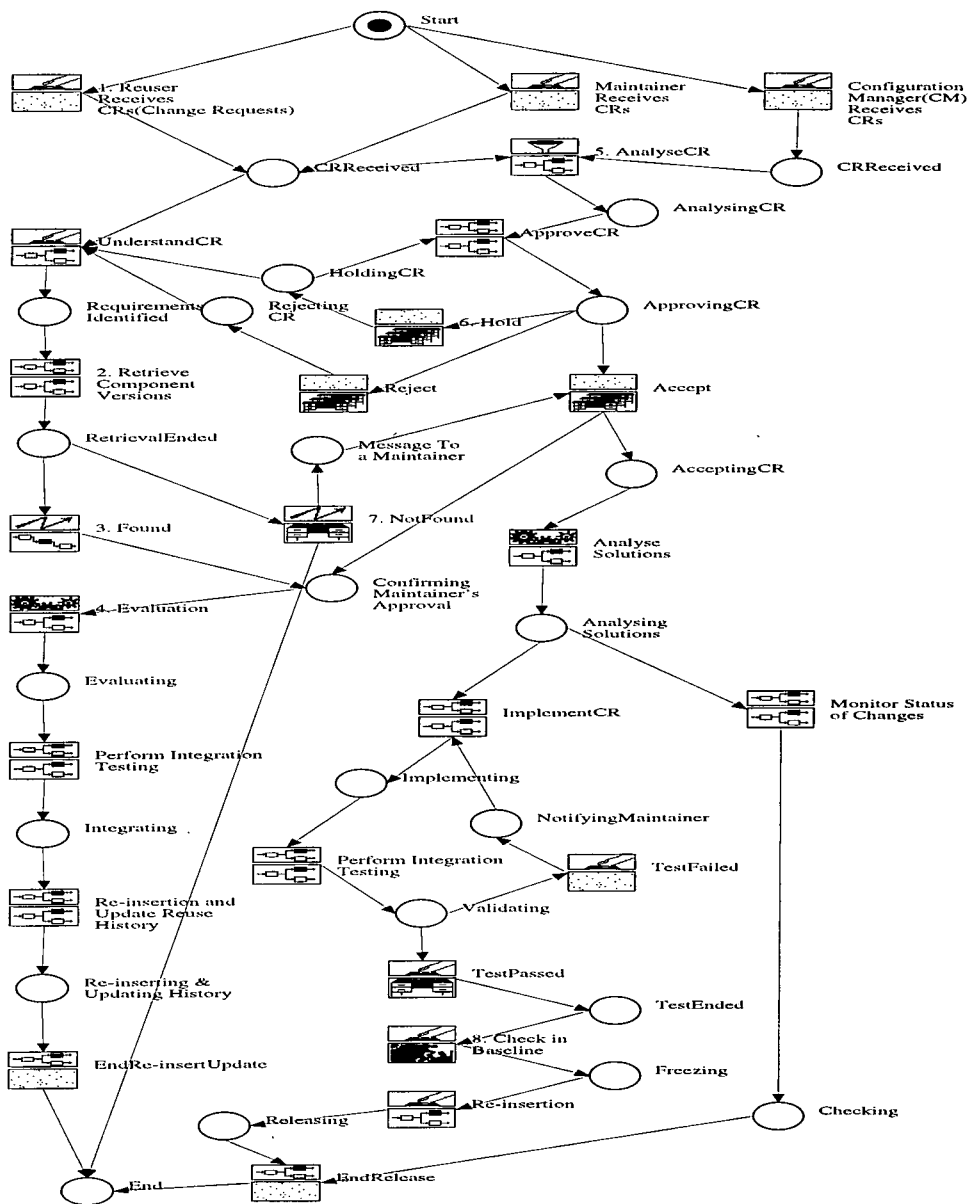
Figure 4.14: The Relationships between the Reuse Process and the Maintenance Process

any changes to an existing SCI. The maintainer implements change requests according to the results of solution and impact analyses. After a maintainer finishes implementing the changes he performs regression testing preceded by integration testing of the new components. After the regression testing is performed successfully by a maintainer, the configuration manager freezes the changed components to set a new baseline and then lets a project manager release a new component. A configuration manager controls and monitors the status of change requests during the implementation of change requests.

Since there exist relationships between the reuse process and maintenance process, the process of 'Maintenance with Reuse (MwR)' can greatly support the process of maintenance if a systematic reuse activity is implemented within a software maintenance environment which is controlled by Software Configuration Management (SCM).

4.3.5 Review of a Product Line and Administration of a Reuse Library

Before populating a reuse library, a librarian/domain manager needs to perform a domain analysis in order to identify *product lines/families* within a development/maintenance organisation. Identification and introduction of product lines are useful for classification of reusable components and building of systems using reusable components.

Concepts Related to a Product Line

The registration of a reusable component should be preceded by classification of the reusable component according to concepts of a *domain analysis* and a *product line*. Cohen, Friedman et al. [25] clarified some terms associated with the product line, and improved greatly product quality and productivity by applying the concept of a

product line to the development and acquisition of electronics systems called C4I for the US Air Force. The relevant terms are discussed below. In particular, we need to clarify the concepts of a product line and a domain as they have some commonalities in some cases.

Component and Asset A *component* is an existing software or document unit that can be used to create a software product. Cohen, Friedman et al. [25] define an *asset* as "a resource or input used to build a product including architectures, tools and COTS (commercial off-the-shelf) software". They had a broad view of an asset encompassing architectures, tools, etc. In this thesis, the author limits the scope of an asset to reusable software components and reusable documents.

Product A *product* is a system delivered to a customer and is composed of a component or an asset. Examples of a product are an inventory control system and an air force monitoring system.

Product Line and Product Family A *product line* is a collection of related systems/products that have specific features and functionalities to meet a common set of customers' requirements whereas a *product family* is a set of related systems that are built from a common set of core assets. Products in the product family are built using these common assets and some system-unique software. Thus, a product line is a market- or customer-driven concept, while a product family is a technology- or implementation-dependent concept [36]. However, the two terms are usually used synonymously. The fill-out form for registration of the prototype is using the term of the product line.

The objective of the product line is to maximise sharing of software resources such as software components, tools, development environments, and test capabilities. The Organisation Domain Modelling (ODM) method [97] facilitates the understanding of product lines. ODM formalises a domain to clarify the domain boundaries by

making clear assumptions about related systems in a domain. It also clarifies the relationships between domains. Product lines are defined through the process of domain modelling (e.g., ODM) that follows the steps below:

1. Categorise the legacy systems developed by an organisation (i.e., building a descriptive model).

2. Identify example programs for each category.

3. Define program relationships and interconnections with other product lines (i.e., building a prescriptive model for each product line). The prescriptive model formally defines the ideal product line and enables the organisation to identify the assets which will be included in each product line.

The last step of the above process uses the results from all the previous steps to produce the prescriptive model for a product line. The prescriptive model formally defines the ideal product line and assists the organisation in identifying and classifying the components related to each product line.

Some examples of product lines are university information processing, traffic control, air-borne air surveillance, air-borne ground surveillance and satellite communications. To facilitate software reuse and rapid prototyping, a product line must be represented by the generic architecture and domain that is depicted below.

Domain and Architecture A *domain* is a specialised body of knowledge and an area of expertise. As the domain is an intangible thing it is unlike a product line or product family which is a specific collection of actual products. An *architecture* is the core asset of a product line. It provides the structure for building products, helps an asset manager to identify product line assets, and defines the means for connecting components.

A *domain analysis* and architecture lay emphasis on understanding the common capabilities of software applications within a product line. A *domain model* is the
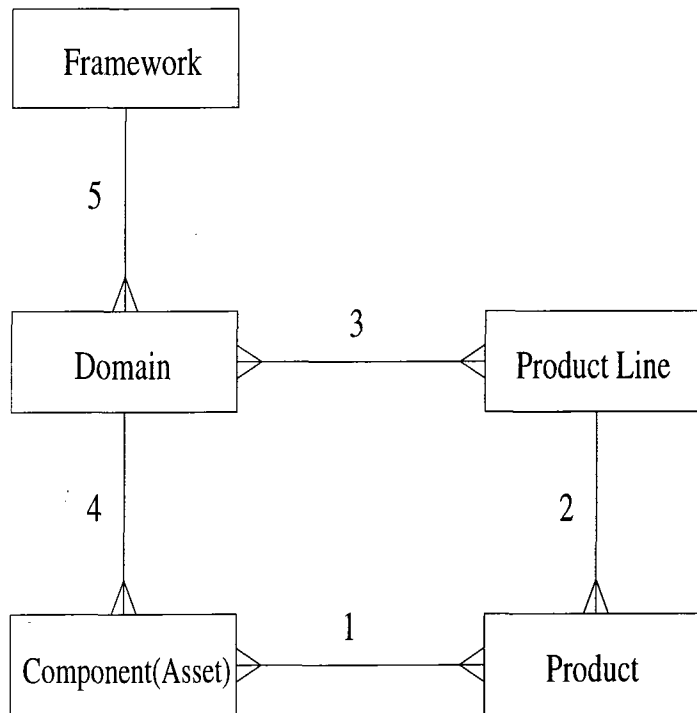
Figure 4.15: Relationships between Concepts Associated with a Product Line

specification of product line assets that defines common capabilities, areas of vari-
ation and reasons for those variations as they are associated with a set of end
users [36].

Figure 4.15 depicts the relationships between some concepts associated with a
product line.

- Number 1 has an 'n to n' relationship. A reusable component can be used to
  build several products. A product consists of several components.

- Number 2 shows an '1 to n' relationship. A product line is composed of
  products that are delivered to a customer. In a few cases, there might be
  an 'n to n' relationship where the development of a system can require an
  integration of products across product lines, using the multiple product lines.

- Number 3 presents an 'n to n' relationship. A domain consists of several
  product lines and a product line can be associated with several domains.

- Number 4 has an '1 to n' relationship. A domain is composed of several assets or components.

- Number 5 shows an '1 to n' relationship. There exist multiple domains in a framework and an architecture.

As shown in the figure 4.15, a product line has a strong relationship with a domain, so the product line should be preceded by a domain analysis.

An application of the product line Although Cohen, Friedman et al. [25] propose 7 criteria as success factors for a product line, the author has addressed a few criteria that can be applied to the development of a product using reusable components. In order for product lines to be successful, the following organisational criteria must be met:

- The process for developing/maintaining an application using the concept of the product line should be built in a development/maintenance environment.

- All the product lines should be defined through the process of domain modelling. The definition of the product line should be performed by domain experts from each product line.

- Each reusable component needs to have related product lines.

- Relationships between products must be clarified. In other words, the reusable component should be linked to relevant products or systems.

- The appropriate alignment of an organisation is essential for an effective application of product lines.

In summary, in terms of software reuse, the product line can be defined as a collection of systems/products that use the common capabilities of a set of reusable components. These common capabilities are identified and specified through the

process of domain analysis. The product line can be an abstraction of system versions and contributes to the population of the reuse library and construction of a new system using reusable components.

Administration of a Reuse Library

Figure 4.1 shows that 'administration of a reuse library' contains three subprocesses, i.e., *'populating a library', 'change control of reusable components', and 'notifying changes''* whose processes have been included in Figure 4.16 that is described in this section.

Reusable components are added to the reuse repository through the populating of a library by a librarian/domain manager. As shown on the left and middle sides of Figure 4.16, the population of a reuse library is performed by a librarian/domain manager in two ways. Firstly, the librarian/domain manager can receive potential reusable components from the reuser or maintainer. Secondly, the librarian/domain manager can also receive the requests for new components from the reuser, maintainer, or supplier.

After quality standards are set up these standards should be referenced and used whenever new components are developed or existing reusable components are maintained. In particular, V & V (Verification and Validation) for reusable components should be introduced to enhance reusability, adaptability, maintainability and portability that are crucial factors for software reuse.

As shown on the left of Figure 4.16, after the librarian/domain manager receives potential components from the reuser and maintainer, he evaluates these components in collaboration with a QA manager. These potential reusable components are component versions that were created as a result of 'white box' reuse. He needs to refer to standards for software quality of his organisation when evaluating possible reusable component versions. After that, according to an organisation's classification scheme, the librarian/domain manager classifies component versions chosen

107

as reusable components in order to register them in a reuse library. Whenever a component version is classified and inserted, the librarian/domain manager should notify the configuration manager, reusers and maintainers in order to propagate what component version was inserted and who inserted it.

A reuser can request potential or new components which he expects to use in the future, and also issue change requests in order to ask for changes to existing reusable components in a reuse library.

As presented in the middle part of Figure 4.16, after the librarian/domain manager receives requests for new components from any reuser, maintainer, or supplier, he decides whether or not requested components have potential reusability in the future. He can accept, reject, or hold these requests for new components. Accepted new components are purchased and inserted into the library after classifying. If he rejects the requests he should notify the requesters of new components. In addition, the librarian/domain manager should propagate the results of decision on these requests to the configuration manager, reusers and maintainers.

The reusable software library may also evolve over time regardless of the type of reuse, i.e., 'black box' reuse, 'white box' reuse and 'gray box' reuse. As shown on the right side of Figure 4.16, after the librarian/domain manager receives change requests (CRs) for reusable components from the reuser and maintainer, he analyses and approves the CRs in collaboration with the CCB. The CCB is in charge of deciding whether CRs for reusable components issued from the reusers and maintainers need to be approved and implemented or not. As in CRs for an existing system, the CRs for the reuse library can be approved, rejected or held by the CCB. When the CRs are implemented the librarian should send the CCB, relevant project managers, maintainers and reusers a message saying that a new version of the reusable component is available.

The procedure to control the CRs for reusable components is similar to the procedure of the CRs for a legacy system. However, there exist a few differences between
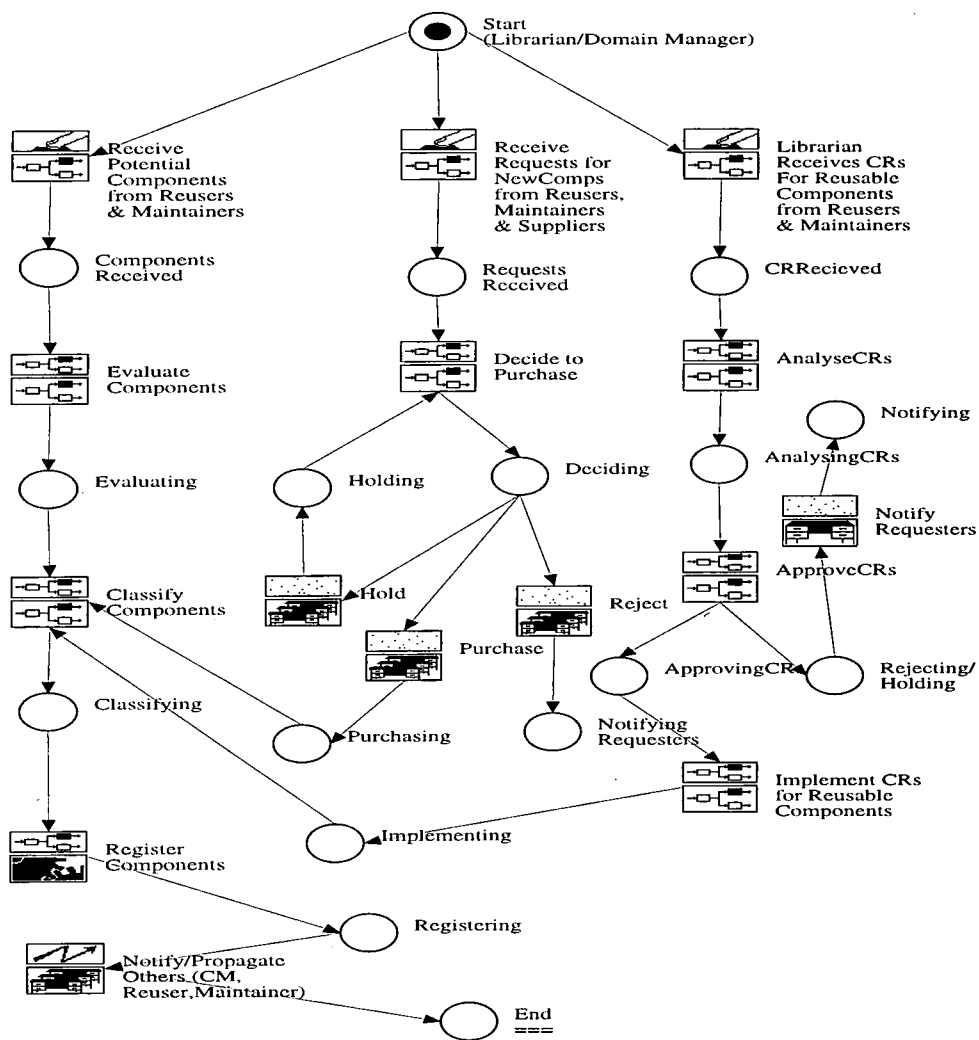
Figure 4.16: The Procedure for Population and Change Control of a Reuse Library

these two kinds of change control. The first difference is that the CCB for change control of reusable components consists of a reuser, a librarian/domain manager, a QA manager and a configuration manager, whereas the CCB for change control of a legacy system includes a maintainer, a configuration manager, a project manager, a QA manager and an end-user but not a librarian/domain manager. The second difference is that the procedure of the change request for the reuse library does not include the step of the "integration testing and regression testing" which is necessary for maintaining a legacy system.

Every reuser, configuration manager and maintainer should be notified whenever reusable components are modified and new versions are created. If a reuser or maintainer discovers a serious error in a reusable component he can ask a librarian/domain manager to delete the component. When deleting a component the librarian/domain manager should notify reusers, the configuration manager, and maintainers about the deletion.

4.4 Summary

The process model of MwR (Maintenance with Reuse) has been produced and refined using Process Weaver that defines activity hierarchies and models the detailed processes for each activity. The activities of the MwR model have been decomposed into four sub-activities which have the models of cooperative procedures. There are two major cooperative procedure models in the MwR model: one is used for controlling evolution of a legacy system through a reuse library; and the other supports the evolution of the reuse repository itself. The former is associated with the tasks of reusers, maintainers, a configuration manager, a project manager and a QA manager whereas the latter is concerned with the roles of a librarian/domain manager, a configuration manager and a QA manager. In order to make a reuse process effective, the reuser and maintainer need to cooperate with each other as the continuity of the reuse and maintenance processes depends on whether or not

there are possible reusable components within a reuse repository.

Although the idea of the product line originated from one of the SCM functionalities (i.e., *product family*), software reuse can benefit from application of the product line to the classification and integration of reusable components. The concept of the product line that enables the optimal sharing of software resources, fits well with the Maintenance with Reuse (MwR) model since the product line may be regarded as a framework of *reusable system versions* that could be customised for other similar system requirements, and it can also be effectively used for classification of reusable components and construction of systems using reusable components.

To enhance the quality of reusable components within a reuse library, a librarian/domain manager must control strictly the population and evolution of the library in collaboration with a QA manager. Potential reusable components for inclusion in the library can come from reusers, maintainers and external suppliers. Change requests (CRs) for reusable components can be issued by reusers and maintainers. If the purchased new components from suppliers and the created new versions from CRs, are registered with the reuse library through the process of Quality Assurance set up by an organisation, the possibility of availability of the reuse library will be increased greatly.

In the following chapter, the implementation of the MwR model is described, including the functions and development of TERRA, TERRA's interaction with SCM, CGI and server, and the description of tools used for implementation of TERRA.

Chapter 5

Implementation of the Model

The model described in the previous chapter has been used as the basis for an experimental implementation. This chapter summarises functionalities of the prototype called *TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library)* and shows the home page of TERRA that provides reusers and maintainers with 5 functions enabling them to deal with changes to both a legacy system and a reuse library. In addition, it addresses TERRA's interaction with SCM, a CGI (Common Gateway Interface) script and a Web server, followed by descriptions of tools such as *free WAIS-sf-2.0.65* and *RCS (Revision Control System)* that have been used for implementation of TERRA on the Unix operating system.

Although the process model of this work contains all activities to support the reuse process, maintenance process and SCM process, all functionalities of the MwR (Maintenance with Reuse) model will not be implemented. This is because the objective of this research is to tackle most problems related to SCM that exists within a reuse process, a maintenance process, and a reuse library itself. In addition, it is impossible to implement the whole part of the model because of time limitations. However, the comprehensive model of MwR can be effectively used for an organisation to build systematic reuse.

5.1 Implementation of TERRA

5.1.1 Functions and Development of TERRA

TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library) is an electronic reuse library that can be accessed through the Internet and is also a tool to manage and control the evolution of reusable components and legacy systems using functions of SCM. The term, 'TERRA' is an acronym of "Tool for Evolution of Reusable and Reconfigurable Assets". It has a meaning of the *earth* or *land* in Latin. The word *Reusable* means the reusable components whereas the word *Reconfigurable* describes that the asset is supported by functionalities of SCM. In addition, the original meaning of TERRA, *earth*, shows it aims at a worldwide reuse server. Thus, TERRA has a implicit meaning of a reuse server that can be accessed by a number of Internet users across the world.

The TERRA prototype for a reuse server has been developed on the WWW in order to enable many people to access and reuse reusable components, to register possible components and to issue a change request to improve the quality of reusable components. The prototype can also be used to build an Intranet application for a specific organisation as well as a general Internet application. The Intranet [65] is an internal network where Internet protocols are used to store and access information and an information repository are front-ended by a web browser such as Netscape or Internet Explorer.

The TERRA prototype developed for the World Wide Web (WWW) has many advantages in terms of portability, traceability, integration with existing tools, and a distributed development/maintenance environment.

It supports most activities of the 'Maintenance with Reuse (DwR)' process that consist of classifying, storing, and retrieving, including controlling changes to reusable components and a legacy system. By providing reusers and maintainers with more information on reuse and maintenance, TERRA helps reusers and maintainers build

113
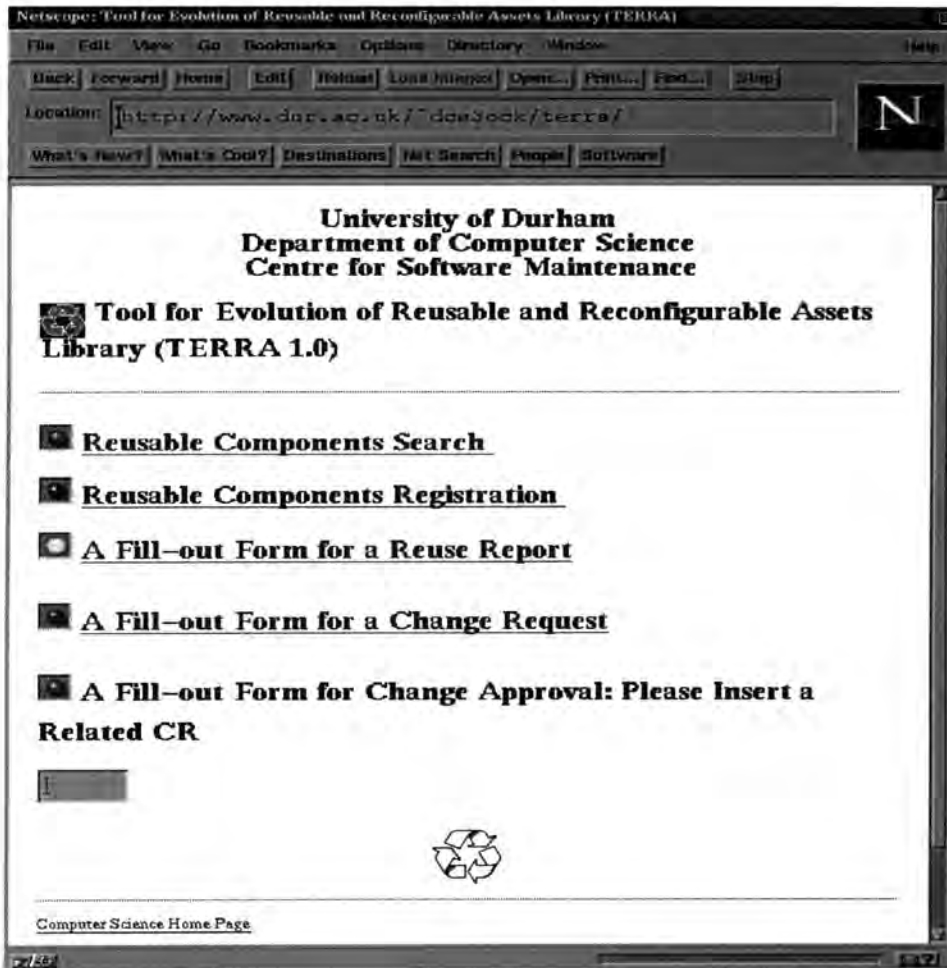
Figure 5.1: Home Page of TERRA

or maintain a system using reusable components. As shown in Figure 5.1, the home page of TERRA provides users with 5 functionalities, i.e., *"Search for Reusable Components, Registration of Reusable Components, A Fill-out Form for a Reuse Report, A Fill-out Form for a Change Request, and A Fill-out Form for Change Approval"*. These functionalities are addressed in detail in Chapter 6.

5.1.2 TERRA's Interaction with SCM, CGI and Server

As shown in Figure 5.2, the HyperText Transfer Protocol (HTTP) is used for transferring HTML forms via the Internet. In other words, HTTP is a method used to transmit data using a hypertext format so that the encoded data can arrive safely
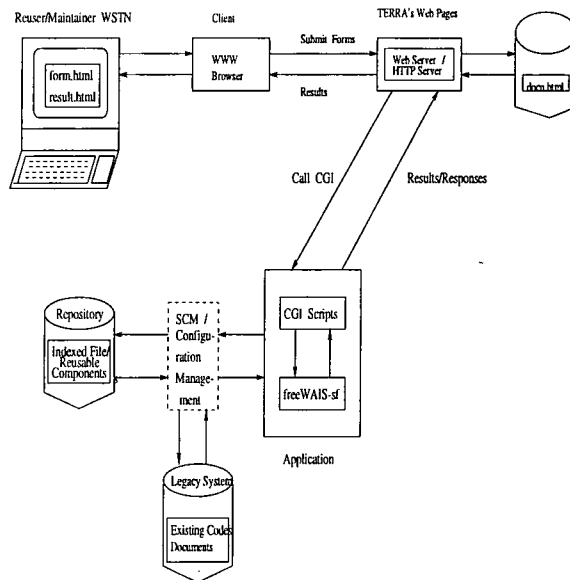
Figure 5.2: TERRA's Interaction with SCM, CGI and Server

at the Web browser. A Web server is a program that receives requests from a Web browser and tries to find the file or program requested [94].

The Common Gateway Interface (CGI) is a mechanism that allows a reuser/user to execute programs on a Web server and to receive their outputs. CGI programs are often used to produce HTML forms on the fly. They are also used to process the input data that a user enters through an HTML form.

In this research, the Perl scripting language has been used to develop CGI applications as Perl is a powerful language that makes it easy to manipulate numbers, text, files, directories, and computer networks. Additionally, it is easy to develop, modify and debug Perl scripts, and Perl has been ported onto many modern operating systems. For this reason, Perl is especially popular with systems programmers and Web developers, but it also has a much broader user community. Perl is no longer used just for text handling and it has grown into a sophisticated and general purpose programming language.

Although Perl was initially designed as a glue language for the Unix system, it can also be run on a variety of other operating systems. Thus, Perl is one of the

most portable programming languages available today [116].

After filling in an HTML form, the sequential process by which a reuser/user
obtains reusable components, has the following steps: Firstly, if a user requests
TERRA's Web page through a Web browser, the Web server returns it to him.
Then, if a user clicks a button after inserting input data for reusable components or
change requests into the Web page, the Web server checks if he has permission to
run the CGI program and that the CGI program also exists. The CGI program is
executed only if the above two conditions are met. The results/responses produced
by the CGI program are transferred to the Web browser through HTTP. Finally,
the Web browser displays the results for reusers/users. Whenever the registration
and retrieval of reusable components are performed, most activities related to SCM
control and management of the related component versions are supported by the
SCM tool shown in the broken line box in Figure 5.2.

5.2   Tools adopted for Implementation of TERRA

5.2.1   freeWAIS-sf-2.0.65

In order to effectively search for reusable components, the tool has adopted 'freeWAIS-
sf-2.0' [88] as a search mechanism, which is currently the most widely used.

How to Index Using freeWAIS-sf-2.0.65

Whenever the reusable components have been registered with a reuse library, the
CGI script indexes the reusable component automatically.

1. Create a directory to keep index files.

2. Create a database file in the form of program sources, text files, dvi files,
   bibtex files, etc.

116

3. Create Makefile, *.fmt and *.fde.

- Makefile : A user needs to change the paths for Macros i.e., WAISINDEX, WAISQ, WAISSEARCH and SWAIS.

- *.fmt : This file is used for format definition. The file is not necessary if the field option '-t fields' is not used. To see the format definition file 'v.fmt' that has been used for this research, refer to Appendix A.2.1.

- *.fde : This file is used for optional format description. The contents of the file are the same as the fields list in a database description file, i.e., '*.src'. The format description file 'v.fde' that has been used for this thesis, is included in Appendix A.2.2.

Instead of using Makefile, we can also use the following commands.

- *waisindex -t fields -d data-base-description-file(i.e. index file) database-file(i.e. source file)* : This command creates 9 index files and other files related to fields. This method might be useful to index reusable codes.

- *waisindex -d index-filename filename filename ....* : This command creates 7 index files. This method seems very simple but it does not provide an efficient search capability as it does not use the field option.

All the index files associated with freeWAIS-sf-2.0.65 are listed in Appendix A.2.3

5.2.2 Revision Control System (RCS)

This research has used the Revision Control System (RCS) [108] as a backbone for version control because this tool has been widely used and is available for a variety of Unix flavours and other platforms. So far, RCS has been adopted as a base for development of several SCM tools.

It has been applied successfully to a variety of development situations that produce documentation, drawings, VLSI layouts, forms letters and articles as well as specifications, test data and source code. RCS automates the storing, retrieval, logging and identification of revisions, and it also provides version selection mechanisms for composing configurations. Comparing it with other tools such as SCCS [93], RCS has a lot of strengths: storing a variety of information, providing a version selection mechanism, managing and merging multiple lines of development, controlling conflicts of coding and access, providing release control, simple user interface, and merging customer's modifications into distributed versions. Detailed functions of RCS [108] can be found in Appendix A.3.

5.3   Summary

The process model of Maintenance with Reuse (MwR) has been implemented on the World Wide Web (WWW) so that the prototype can provide reusers and maintainers with a platform independent, distributed maintenance environment. The TERRA prototype has good flexibility that can be integrated with existing tools on a Unix system. In particular, Perl has been used to implement CGI applications for the prototype as it is one of the best portable scripting languages. If access control is provided using a user ID and password, TERRA can be effectively used for building Intranet applications as well as Internet applications.

The reuser, maintainer, configuration manager, librarian, end user, etc. can be assigned the five fill-out forms of the TERRA home page through access control. For instance, the reuser and maintainer can have access to the field, "Reusable Components Search" in order to implement change requests (CRs), and the field, "A Fill-out Form for a Reuse Report" to record their experiences with reusing software components. The librarian can access the field "Reusable Components Registration" in order to put reusable components into a reuse repository. In addition, the fields "A Fill-out Form for a Change Request" and "A Fill-out Form for Change Approval"

can be accessed by the reuser and maintainer, and the configuration manager and domain manager, respectively.

In the next chapter, a detailed description for the TERRA prototype use is given, focusing on registration of reusable components, retrieval of reusable components, recording the history of reuse, filling out a change request (CR) form, and filling in a change approval form.

Chapter 6

Operation of the TERRA prototype

This chapter describes the procedures of operation of TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library) that has been developed to support the MwR (Maintenance with Reuse) Model. In order to show how to use the TERRA prototype in this chapter, the author will not enter sample data into all fill out forms as use of the prototype will be fully illustrated in the case study using real data in Chapter 7. The order of presentation will be based on the sequence of activities shown in Figure 5.1 of Chapter 5.

In Section 6.1, the steps for registration of reusable components are described, giving a brief description for the entry fields of the fill-out form. Section 6.2 shows how to retrieve a reusable component using the search mechanism, and also presents the search results format and a detailed specification of a retrieved reusable component, that contains "the history of change" and "the history of reuse" on the bottom of the specification both of which are very useful to reusers and maintainers. In Section 6.3, a fill-out form for the reuse history is described. Section 6.4 describes the entry form used to enter change requests (CRs) which are issued in order to maintain a legacy system and a reuse library. Finally, Section 6.5 shows a

120

fill-in form that the CCB uses in order to decide if the CRs are approved, rejected or held.

6.1 Registration of Reusable Components

As shown in Figure A.1 of Appendix A, two fill-out forms are used to insert the two types of reusable components, i.e., software and documents. Figures 6.1, 6.2 and 6.3 show entry fields for the registration of reusable software. The identifier of the reusable software component is numbered in the form 'S0001' whereas the reusable document is named in the form 'R0001'. These identifiers are automatically generated and displayed, and can be changed by users. Input items that must be inserted by users are: component IDs, component name, author name, date of creation, date of registration, maintainer's name, operating system, computer language, component format, related domain, related methods and techniques, and keywords.

A librarian/domain manager is responsible for registering reusable components and the sources of these components come from an existing system, external suppliers or new component versions created as a result of change requests (i.e., 'white box' reuse). The entry fields "operating system" and "computer language" are related only to reusable software. The "component format" entry field for reusable software has the options of source code, executable code or data file. On the contrary, the "component format" entry field for registration of reusable documents can be chosen from plain text, postscript or HTML. The "related domain", and "related method and technique" entry fields have 4 choices and 33 options, respectively. To enhance the effect of the search mechanism, keywords as well as all the fields identified as 'required', should be chosen properly and input.

Without a list of links from reusable components to every product using them, a change or a fix of a component cannot be managed or promulgated. The entry

Figure 6.1: A Fill-out Form for Reusable Components' Registration: Part #1

File   Edit   View   Go   Bookmarks   Options   Directory   Window   Help

Back Forward Home   Edit   Reload Load Images Open... Print... Find...   Stop

Location: http://hexham/cgi-bin/dce3ock/reg_soft_doc.pl?s

What's New? What's Cool? Destinations Net Search People Software

Supplier/Registerer Name

Abstract/Summary

Component Size

KBytes

Operating Systems (Required)

Computer Language (Required)

Component Format (Required) --- Please choose one of the followings.

    Source Code
    Executable Code
    Data File

Related Domain (Required) --- Please choose one of the followings.

    Business Data Processing (BDP)
    Telecommunication
    Aerospace

Figure 6.2: A Fill-out Form for Reusable Components' Registration: Part #2

123

Figure 6.3: A Fill-out Form for Reusable Components' Registration: Part #3
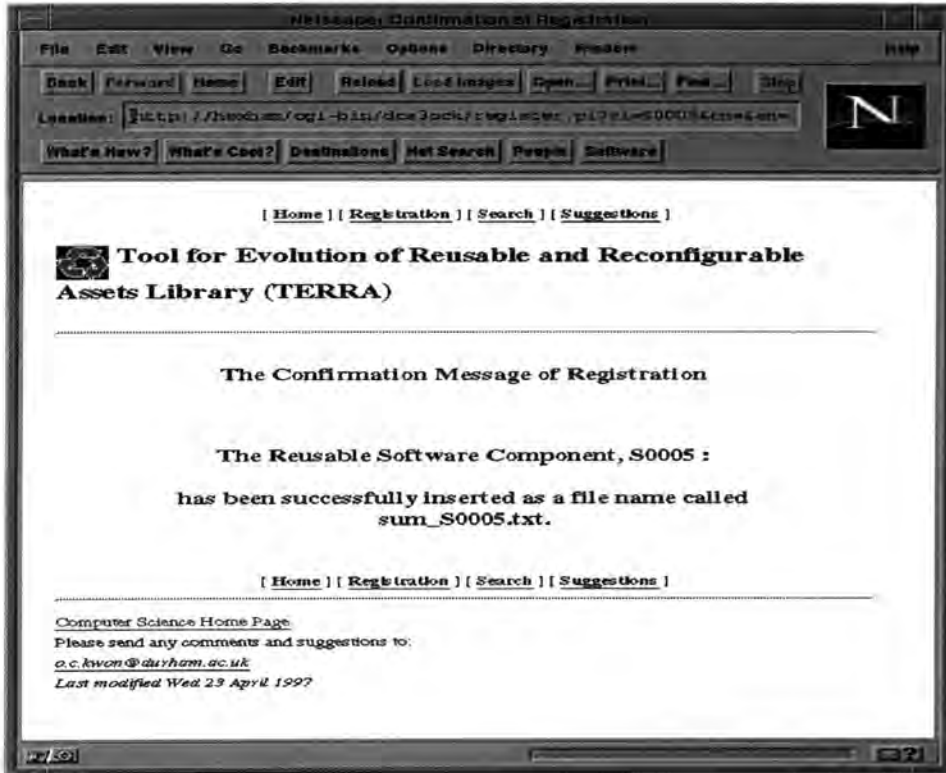
Figure 6.4: A Message for Confirmation of Components' Registration

fields "related products" and "related product lines" can be used to control and promulgate changes to reusable components within a reuse library. A librarian/domain manager who is responsible for managing a reusable components library, should maintain as many potential and useful versions of components as possible.

The field "related components" is useful for impact analysis and regression testing. Users can refer to the online help for understanding of these terms. A "downloadable file" is an actual file name stored in a reuse repository, which can be downloaded by a reuser after retrieval. If the user finishes inserting every mandatory field and clicks on the 'submit button', then he will get the message for confirmation of registration as shown in Figure 6.4. Whenever a reusable component is inserted into the reuse repository, indexing for the search mechanism is automatically performed by using the information in the related files that are already defined for indexing.
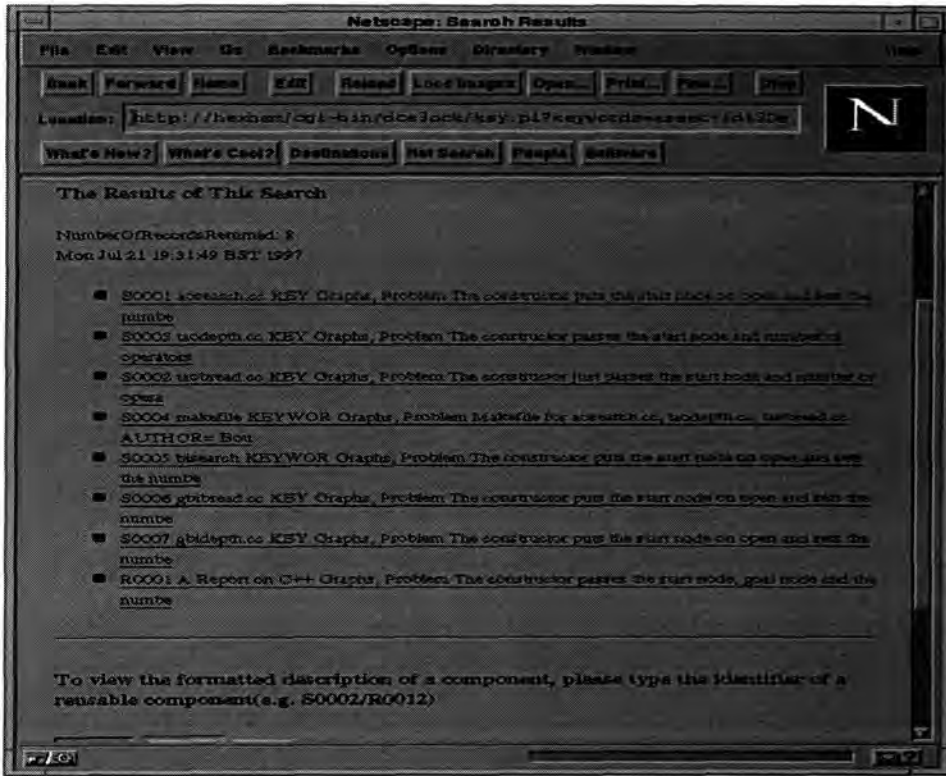
Figure 6.5: Reusable Components Retrieved by a Search Mechanism

6.2 Search for Reusable Components

Figure 6.5 lists the reusable components retrieved by keyword search whose field option for the search was 'ci=S0001', as shown in Figure A.2 of Appendix A. Free text can be used to search for reusable components and there are also several field options as follows: *ci(identifier of a reusable component), cn(name of a reusable component), an(author name), os(operating system), cl(computer language), cf(component format), dm(domain), mt(method and technique),* and *kw(keyword).*

A brief description of each component is displayed according to the format of a headline that is defined in an index file. The format for this search is as follows: *ci(Component ID), cn(Component Name), an(Author Name), os(Operating Systems), cl(Computer Language)* and *cf(Component-Format).* The reusable components are sorted and displayed depending on the score for each component that is assigned by a search engine. The reuser can read a brief summary of the reusable
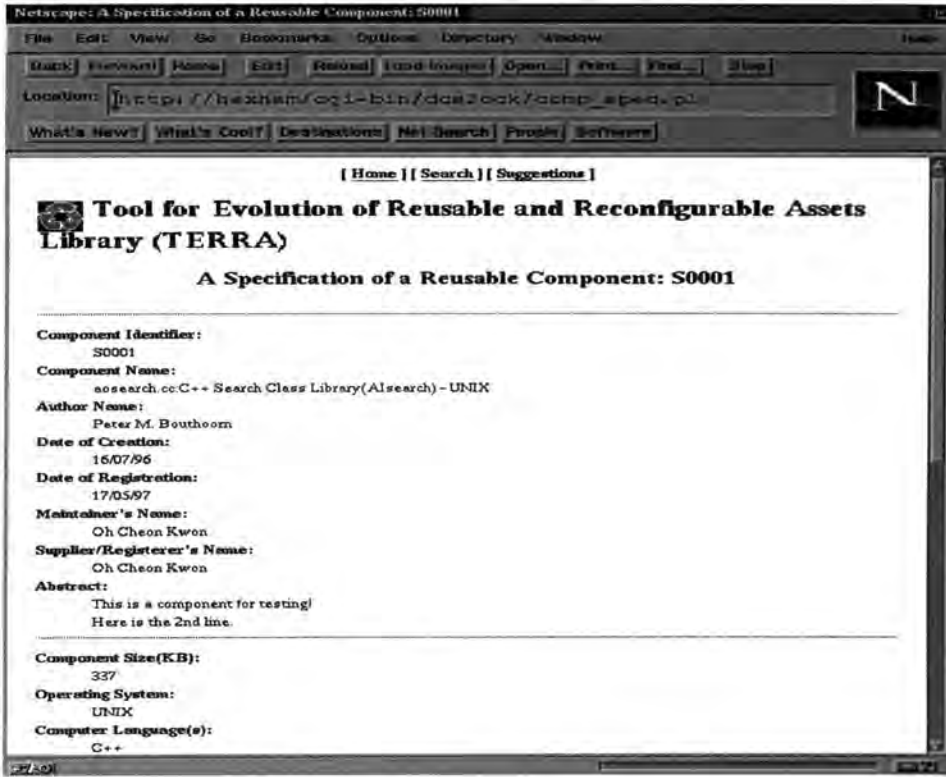
Figure 6.6: A Description of a Retrieved Reusable Component: Part #1

component by clicking on the HREF links returned. He can also view the formatted specification of the reusable component by inserting the component's identifier into the input field on the bottom of the above figure.

Figures 6.6 and 6.7 show formatted and detailed information on a reusable component retrieved by TERRA's search mechanism. The description is created from the file information stored in a repository by a Common Gateway Interface (CGI) script on the fly. After checking the detailed component information, the reuser can download the latest version of the reusable asset. He can also retrieve the latest version through the change history that will be described in the following two paragraphs. The description part of a reusable component is followed by some information on the "history of change", the "history of reuse", the "evaluation of reusers" and the "tree of versions" which are very useful for reusers. If reusers are interested in the retrieved component they may want to view this information.

Netscape: A Specification of a Reusable Component: S0001

File   Edit   View   Go   Bookmarks   Options   Directory   Window   Help

Back Forward Home   Edit   Reload Load Images Open... Print... Find...   Stop

Location: http://beaham/cgi-bin/docbook/comp_spec.pl

What's New? What's Cool? Destinations Net Search People Software

Computer Language(s):
    C++

Component Format:
    Source Code
Related Domain:
    Business Data Processing (BDP)
Related Methods and Techniques:
    Artificial Intelligence

Key Words:
    C++ CLASS LIBRARY, PROBLEM SOLVING, SEARCH ALGORITHMS
Related Components:
    R0011: The Report on C++ SEARCH CLASS LIBRARY (AISEARCH)
Related Products:
    Product #1, Product #2
Related Product Lines:
    PL #3, PL #4
Downloadable File(s):
    Retrieve the Current Version: aosearch.cc

View the History of Change
View the History of Reuse
View the Evaluation of Reusers
View the Tree of Versions

[Home] [Search] [Suggestions]

Computer Science Home Page
Please send any comments and suggestions to:
o.c.kwon@durham.ac.uk
Last modified Mon May 19 1997

Figure 6.7: A Description of a Retrieved Reusable Component: Part #2

The change history displays differences between revisions associated with a down-loadable file in which the reuser is interested. Figure 6.8 shows the change history of a reusable component, 'S0001(aosearch.cc)', that is, a summary of changes made between revision 1.1 and revision 1.6. The reuser needs to compare several revisions so that he can choose a specific version that matches the requirements of a change request (CR) for a legacy system exactly or very closely. He can also insert the version numbers of two revisions in order to know exactly what changes have been made in the two revisions, and to know differences between them. The entry fields for the two version numbers can be found at the bottom of Figure 6.8. Figure A.3 in Appendix A is used for the entry of two version numbers, and Figure A.4 shows "the difference list of two revisions" extracted using the entry of two version numbers shown in Figure A.3.

After the reuser compares two versions of a component he needs to enter a version number to extract a specific revision of the reusable component. The activity

Figure 6.8: The History of Changes made to a Reusable Component

of evaluation by a reuser consists of understanding retrieved component versions, investigating alternatives to meet the requirements of reuse, and selecting a better component version for reuse. After evaluating reusable component versions, a reuser has to adopt either 'white box' reuse or 'black box' reuse depending on the level of adaptation/modification of retrieved reusable component versions that is required. Only if the estimated effort required to modify a reusable component version from the repository does not surpass the effort to adapt an existing legacy component, the reuser will adopt 'white box' reuse. If the specific revision needs to be modified for 'white box' reuse he follows a process of software maintenance. After modifying a reusable component, the reuser performs a unit test.

A better Syntactic Interconnection Language such as SySL (System Structure Language) [105] and PCL (Proteus Configuration Language) [101] can be used to construct composition rules for an integration test. If the integration test is performed successfully the reuser updates files related to reuse experiences (e.g., a reuse

history file and a evaluation file).

The reuser can refer to the experience report from other reusers by clicking on 'View the History of Reuse'. This HREF link shows some information on the reuse type, the summary of modifications, the degree of modifications, the related domain, the related product line, the related product, etc. Section 6.3 presents a fill-out form for the entry of the reuse history. Figure A.5 in the Appendix shows one example of a reuser's experience report. The accumulated experience report from reusers encourages other potential reusers to further reuse the reusable components.

This document adopted a 'POST' operation as a request method since the form data created using the 'POST' method will have automatically expired from the cache, so the reuser has to repost the form data to recreate the document by pressing the reload button. In addition, the reuser can always click on links to "the history of change", "the history of reuse", "the evaluation of reusers", and "the tree of versions" in order to view the current information. To allow this functionality, the current time is assigned to a variable and the variable is appended to the end of an HREF link so that the HREF link can be interpreted as the different version (i.e., different link) and the Web browser can reload the file related to the HREF link.

6.3 Report on the Reuse History

A reuser's experience needs to be entered to be used as data for "the history of reuse" shown in Figure 6.7. As shown in Figure 6.9, a fill-out form consists of component ID, component version, reuser name, date of creation, reuse type, summary of modifications, amount of modification, domain, related product line, related product, and other comments. The mandatory fields are component ID, component version, reuser name, date of creation, reuse type, domain, related product line, and related product. The other input fields not shown in Figure 6.9 are included in Figure 7.10 of Chapter 7.

130
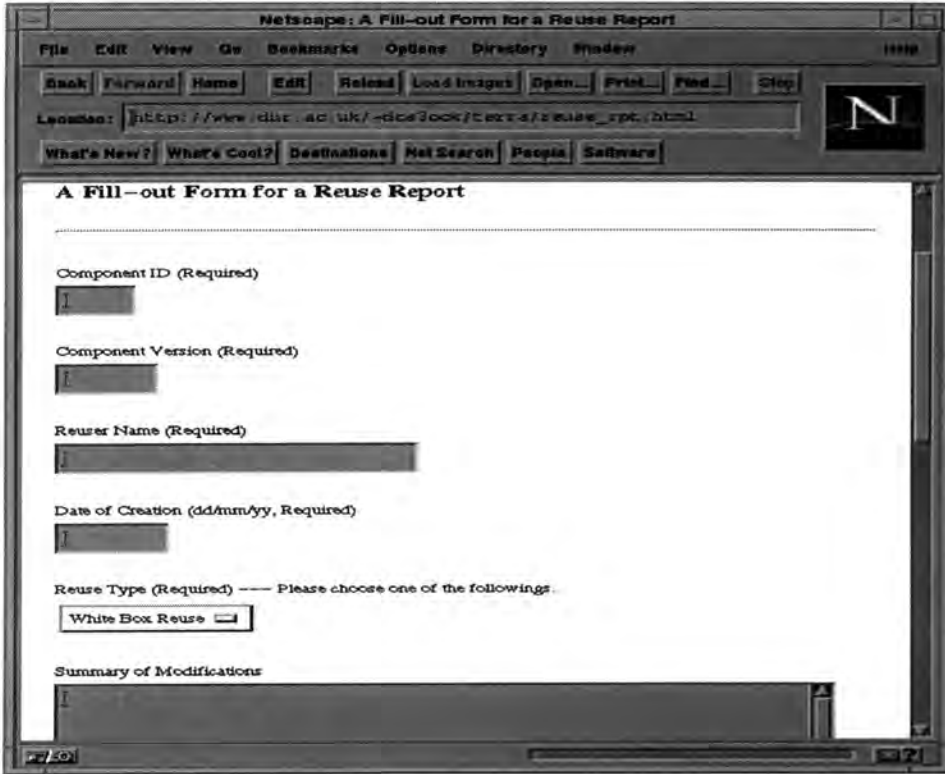
Figure 6.9: A Fill-out Form for a Reuse Report

The reuse type can be chosen from one of 'white box' reuse, 'black box' reuse and 'gray box' reuse. The default value of the reuse type field is 'white box' reuse. If the reuser needs to modify a reusable component before reuse, he can reference summary of modifications. There are 6 domains: Business Data Processing, Telecommunication, Aerospace, Chemistry, Military Industry and Mechanical Engineering. After the reuser fills out the form he submits his reuse report and gets a confirmation message.

6.4 An Entry Form for a Change Request

A maintainer and project manager analyse a change request (CR) for a legacy system whereas a librarian/domain manager reviews a CR for a reuse library. They decide if a CR comes from an incorrect operation, incorrect documentation, or incor-

rect software. Then, they produce a change proposal/Engineering Change Proposal (ECP) according to SCM practices which will be submitted to the CCB (Configuration Control Board). However, this prototype excludes the entry of the ECP since a fill-out form for a CR contains some contents of the ECP. The ECP can be classified as either *emergency, urgent, or routine.*

Both a configuration manager and a librarian/domain manager need to cooperate with each other when dealing with change requests. The configuration manager and librarian/domain manager are responsible for monitoring the implementation of all change requests for an existing system and a reuse library, respectively. The CCB is responsible for approving, disapproving/rejecting, or holding a submitted change proposal.

As shown in Figure 6.10, a fill-out form for a change request consists of the fields: change request number, originator name, date of issue, type of change request, related system, related component ID, related component version, type of maintenance, status of change request, description of change, and reason for change. Mandatory entry fields are change request no, originator name, date of issue, type of change request, status of change request. The whole fill-out form for a change request is shown in Figure 7.1 of Chapter 7.

The format of "a change request number" is like 'CR999' and is generated automatically in a continuous sequence. The "an originator" field must be recorded as this person should be informed of the progress of the CR implementation. The originator can be a reuser, maintainer or end-user. There are 3 types of change request: an existing system, a reuse library and a production/static library. The TERRA prototype supports functionalities for SCM to control changes over a production library as well as a legacy system and a reuse library. The production/static library is a repository where components of a system already produced are frozen and stored before release. "A related system" in the case of change requests for reusable components should be the name of a reuse library. However, the related system in other types of CRs can be identified and entered. If "a related component ID and ver-

File   Edit   View   Go   Bookmarks   Options   Directory   Window   Help

Back   Forward   Home   Edit   Reload   Load Images   Open...   Print...   Find...   Stop

Location: http://hexham/cgi-bin/doc3ock/change_req.pl

What's New?   What's Cool?   Destinations   Net Search   People   Software

A Fill—out Form for a Change Request

Change Request No (Required)

CR016   **CR016**

Originator Name (Required)

Gary Smith

Date of Issue (dd/mm/yy, Required)

19/8/97

Type of a Change Request (Required) --- Please choose one of the followings.

An Existing System
A Reusable Library
A Production/Static Library

Related System (If applicable)

XYZ System

Related Component ID (If applicable)

Figure 6.10: A Fill-out Form for a Change Request

sion" can be identified the originator of a change request inserts them into the form. The maintenance type can be one of the following: *perfective maintenance, adaptive maintenance, corrective maintenance, and preventive maintenance.*

The default value of the maintenance type is "perfective maintenance", i.e., the modification of a software product after delivery in order to enhance performance or to meet new user requirements. "Adaptive maintenance" is the modification of a software product after delivery to enable a software system to adapt to a different environment. "Corrective maintenance" is the modification of a software product after delivery to fix errors discovered during system operation. "Preventive maintenance" is the modification of a software product after delivery to prevent problems before they occur. For example, the modification of a software product to increase maintainability is a form of "preventive maintenance".

The status of a change request can be one of *issued, approved, held* and *rejected.*

The originator enters a description of the change and the reason for the change in order to describe why the change has been proposed. After filling in the other required fields, he submits the change request. He then gets the message for confirmation of a change request entry, which says "The Change Request CR999: has been successfully inserted as a file name called CR999.txt".

6.5　An Entry Form for Change Approval

After the librarian/domain manager receives change requests for reusable components from the reuser and maintainer, he analyses and approves the change requests in collaboration with the CCB. When the change request is implemented the librarian/domain manager and configuration manager should send the CCB, maintainers and reusers a message saying that a new version of the component is available. The procedure of the change requests (CRs) for a legacy system is similar to the procedure to control the CRs for reusable components. However, there exist a few differences between these two kinds of change control as shown in Section 4.3.5 in Chapter 4.

In order to display a fill-out form for change approval from the home page of TERRA, a CCB member needs to insert a related change request number. If the change request number is valid then a fill-out form for change approval is displayed, which consists of three frames. As shown in Figure 6.11, every frame initially acts independently, depending on the user's intention. The top left frame displays some information about the change request that has been inserted by a user. The top right frame shows a summary of all the change requests which are stored in the repository. If the user clicks on a specific change request, he can view more details about the change request from the top left frame. In general, change approval is associated with several change requests, so the user needs to reference some change requests while he enters the information into the fill-out form.
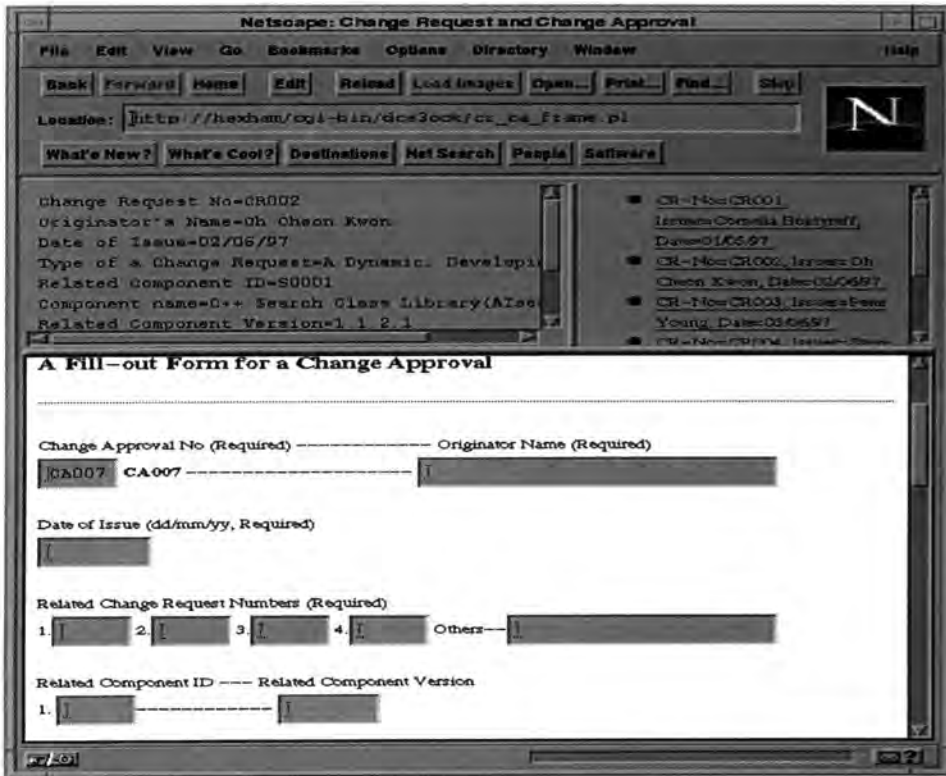
Figure 6.11: A Fill-out Form for Change Approval

The fill-out form for change approval consists of the fields: change approval number, originator name, date of issue, related change request numbers, related legacy system, related component ID, related component version, related product ID, related product line, type of maintenance, status of change request, specification of change, due date of change, estimated manpower to implement the change, and estimated cost for change. The required fields include change approval number, date of issue, related change request numbers, related component ID and version, and status of the change request. The other entry fields not shown in Figure 6.11 can be found in Figure 7.8 of Chapter 7.

A "change approval number" is automatically displayed and can be updated by a number that the user inserts. The user, who is a member of the CCB, can insert several "related change requests" after viewing change requests using the top two frames of Figure 6.11. Even though there exist entry fields for the related product lines and products that have used reusable components, he can also enter the name

of a related legacy system that the CR came from. After performing an impact analysis of the change the user enters component IDs and versions associated with changes to be implemented. The user can enter one out of four types of maintenance if applicable.

A simple description of the change is given in the field, "specification of change". A more detailed specification can be appended to the change approval form. The estimated manpower and implementation cost that are calculated by a project manager, maintainer and domain manager who are members of the CCB, can be input using units of man month and pound.

The status of a change request contains 4 options which are exactly the same as the form for a CR. After the CCB reviews and evaluates the contents of the change approval form, the CCB can decide whether the CR should be approved, rejected, or held. The criteria used by the CCB when judging the CR, depend on the cost and benefit of the change, the extent of the change, and the importance and urgency of the change.

If the user finishes inserting all the mandatory fields and clicks on the 'submit button', then he will get the message for confirmation of the change approval entry, such as "The Change Approval, CA007, has been successfully inserted as a file name called CA007.txt".

6.6 Summary

In this chapter, the use of the TERRA prototype that has been developed to support the Maintenance with Reuse (MwR) model, has been described using a simple description for the entry fields of the fill-out forms rather than using real data. Most of the fill-out forms for registration, retrieval, report, change request and change approval have been described, but some of them have been included in Appendix A to lessen the complexity of this chapter.

136

The results from a trial operation have been used to fix errors and to improve the functionality of the prototype. In particular, the fill-out form for registration of reusable components has been divided into two separate forms, i.e., software and documentation in order to reduce overload that might result from accesses of a number of users, although these two fill-out forms are similar to each other. Whenever users submit fill-out forms to a reuse server, they receive confirmation messages from the server that the submission has been carried out successfully. In the first prototype, the files within an HTML form that was produced on the fly by Perl scripts, did not show current versions by user's clicking on URLs in displayed forms even though files related to the URLs had been updated. In order to solve this problem, a POST method has been adopted as a request method for forms, and a variable for holding the current time whenever users click on an HREF link is appended to the link. In the next chapter, the case study based on 4 scenarios is carried out using real data, and the results of the case study are discussed.

Chapter 7

Scenario Based Case Study

7.1 Introduction

The objective of this case study is to evaluate a MwR (Maintenance with Reuse) model by using TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library), and to improve and upgrade the associated method and tool developed during this research work. In order to demonstrate the capabilities of the TERRA prototype, this case study is applied to several problem areas for software reuse: cataloguing reusable components, retrieving them from a repository efficiently, controlling the changes to reusable components, propagating the changes of a component to the reusers of the component, and controlling the changes to a legacy system through the process of reuse. The composition of a system using reusable components is one of problems with reuse, but it has not been implemented in the TERRA prototype because of time limitations although the functionality is included in the MwR model.

This case study does not address all the concerns of Software Configuration Management (SCM), but concentrates on demonstrating the practical use of TERRA and evaluating its model. The objectives of the case study are to:

- Show how the TERRA prototype developed in this research may be used in practice. The steps to introduce the TERRA prototype into Company XYZ will be presented.

- Evaluate a MwR (Maintenance with Reuse) model. The goal of the prototype is to support SCM functionalities for a reuse library within a software maintenance environment.

- Discuss specific aspects of the model and tool.

- Refine a reuse process and a maintenance process using results obtained from this case study.

This section is organised as follows. Section 7.2 presents the Company XYZ's problems associated with software reuse and maintenance based on a C++ GUI Framework called 'V' that is summarised in Section 7.3.1. Section 7.4 gives detailed descriptions of the major steps for four scenarios enacted in this case study. Finally, Section 7.6 discusses the case study and its implications.

7.2 Problem Statement for the Case Study

Company XYZ[1] has developed a C++ GUI framework for Windows and X, which is called V. The V framework has been customised for over 100 sites and ported to 10 platforms, so a lot of change requests have been issued and controlling of changes to existing components has been required.

The V system consists of many components (i.e., more than 550 files), so Company XYZ has recognised the necessity of Software Configuration Management (SCM) in

---

[1]In reality, V was developed by Professor B. Wampler at the University of New Mexico, but the author assumes, for the purposes of this scenario, that the development and maintenance of V are being carried out by Company XYZ. The author would like to thank Prof. Bruce Wampler for his cooperation in providing valuable source code and documents of the V framework.

order to efficiently maintain various evolutions of the legacy system. In addition, some components, for example, the "vApp class", serve as the base classes for building applications. There must be exactly one instance of an object derived from each base class. Thus, the base class needs to be maintained as a reusable component. Each instance of an object can also be registered with a reuse library. Although Company XYZ has seen software reuse as a solution to reduce cost and to increase quality and timeliness, it has not found a better reuse library to catalogue and retrieve reusable components, to control the changes to reusable components, and to manage versions created as a result of 'white box' reuse.

7.3   Preparation for the Case Study

7.3.1   Data for the Case Study

V^2, which is a "Freeware Portable C++ GUI Framework for Windows and X", has been used for the case study. V is an easy-to-program and cross-platform C++ GUI Framework which provides the easiest way to write C++ Graphical User Interface (GUI) applications which are available as commercial, shareware, or freeware. V is available under the terms of the GNU Library General Public Licence, so it will remain freely available to the public.

The V framework has been used during several semesters for the software engineering class at the University of New Mexico, and has proven to be extremely easy to learn and use, as well as reliable for commercial software. Most standard GUI objects are supported by V, including windows with menus, status bars, tool bars, and a drawing canvas; modal and modeless dialogs with the most common controls (buttons, lists, labels, text entry, check and radio buttons, etc.); and portable printing support.

---

[2]The V framework can be found at http://www.cs.unm.edu/%7Ewampler/vgui/vgui.html (valid on 1 August 1997).

The V system consists of 551 components including documentation in LaTeX format as well as C++ sources for Windows and X. All the components of the V system are kept in a repository supported by Unix's Revision Control System (RCS). The class, "vApp.cxx", has been chosen as a reusable component since the component has been used for a base class and the size of the component is rather big i.e., 792 LOC (Lines of Code), so the file can be classified as a component to which many changes have occurred.

7.3.2 Storing the V Legacy System into a RCS Repository

All the files of the V legacy system should be stored in the production library that can be supported by an SCM environment. Thus, the first step is to check the V system into the RCS (Revision Control System) [108] environment. Although the Company XYZ has kept 5 old system versions the company has not performed any change control and version control efficiently. The organisation has only used the concept of a system version, not a component version. The descriptions of system versions have been maintained in a user's manual. The records of change requests (CRs) are manually kept until the CRs are implemented by a maintainer.

7.4 Scenarios for the Case Study

The procedure of change control starts with the issue of a change request and ends with the implementation or rejection of the change request. There are three types of change requests (CRs): a change request for a legacy system, a change request for a reusable component, and a change request for a static library (i.e., production library). As the change control procedure of a change request for the production library within a development environment is similar to that of a change request for a legacy system, and this research is based on a maintenance environment, the case study will only deal with change requests for a legacy system and a reuse library.

A librarian or domain manager is responsible for managing evolution of a reuse library. Even a library that supports 'black box' reuse is subject to change over time. Whenever reusable components within a reuse library have changed the librarian/domain manager should inform a configuration manager, a reuser and a maintainer that the contents of the library have been updated.

A maintainer is in charge of managing changes to an existing system. The maintainer is encouraged to use possible reusable components within a reuse library in order to implement approved change requests (CRs). To do so, the maintainer needs to cooperate with a reuser. Thus, this case study focusses on two sources of CRs, i.e., changes for a reuse library and an existing system. The procedure for introducing the TERRA prototype into Company XYZ is described within the four particular scenarios we enacted. These scenarios will form the basis for the evaluation in the next Chapter and they are as follows.

- Scenario # 1: Populating a Reuse Repository.
  This scenario shows how the librarian/domain manager registers the specification of a reusable component with an RCS repository and includes the librarian's role in storing the file of the reusable component into an RCS repository.

- Scenario # 2: Procedure of Change Control for an Existing System.
  This scenario describes the procedure to implement the CR for a legacy system, which can be supported by the process of maintenance.

- Scenario # 3: Procedure of Combined Change Requests for a Reusable Component and an Existing Component.
  This scenario shows an example of integration of two CRs that have been issued from a librarian and a maintainer, respectively.

- Scenario # 4: Procedure of Change Control for an Existing System.
  This scenario presents the procedure to implement the CR for a legacy system. The CR can be implemented by the process of reuse as the reuser can find out the candidate reusable component from the reuse library.

142

7.4.1 Scenario # 1: Populating a Reuse Repository

A librarian/domain manager is responsible for registering the specifications of reusable components with a reuse repository. As shown in Figure 6.1 of Chapter 6, the specification of a reusable component is stored in the form 'sum_S0001.txt' by using a fill-out form for component registration.

The librarian/domain manager of Company XYZ has decided to register a C++ class, 'vapp.cxx', and related components as he recognised that the component has been frequently used for a base class to create the instance of an object over the past few years.

Creating a Directory to Keep Index Files

In order to effectively retrieve reusable components, the TERRA prototype has adopted 'freeWAIS-sf-2.0' [88] as a search mechanism, which is currently one of most powerful search engines. Whenever reusable components are registered, all the files associated with indexing are automatically updated.

Creating the Files Related to Indexing, 'v.fmt and v.fde'

The 'v.fmt' file is used to define the format of indexing. For details about 'v.fmt' see Appendix A.2. The 'v.fde' file contains an optional format description. This is a plain text file whose contents are also added to the database description file 'v.src'. Thus, the contents of the file 'v.fde' are the same as fields list in the file 'v.src'.

Registering the Specifications of the Reusable Component

Reusable components come from two sources, i.e., an existing system and an external supplier. A reuser and a maintainer can submit potential reusable components to a librarian/domain manager. The librarian/domain manager needs to evaluate

143

these components in collaboration with a quality manager. Accepted reusable components should be registered with a reuse repository by using a fill-out form as shown in Figure 6.1 in Chapter 6. A maintainer, "Peter Anderson", asked the librarian to evaluate whether or not a component "vapp.cxx version 1.1" can become a reusable component. The librarian/domain manager decided to accept the component as a reusable asset. As shown in Figure 4.16 of Chapter 4, the component received from a maintainer is evaluated, classified and put into a reuse library by the librarian/domain manager.

To demonstrate how the TERRA prototype works in a maintenance environment, several specifications of reusable components have been registered with a repository for reusable components within TERRA. The reusable software and documentation are stored in two separate directories. The evolution of these sources is controlled by RCS. The specifications of reusable components registered in a library are as follows:

- vapp.cxx: The "vApp" class serves as the top level class used to build an application. There must be exactly one instance of an object derived from the "vApp" class. The base class contains and hides the code for interacting with the host windowing system, and serves to simplify using the windowing system.

- vapp.cxx related components: vapp.h, vwindow.h, vcmdwin.h, vctlclrs.h, vcolor.h, stdio.h, stdlib.h, vrefch4.tex, vcmdwin.cxx, vawinfo.cxx.

- vawinfo.cxx: This is a utility class to interface views to models. This class is intended to be used as a base class for deriving the user's own "myAppWin-Info" class. The purpose of such a class is to serve as a controller data base for the MVC architecture.

- vawinfo.cxx related components: vawinfo.h, string.h, vapp.cxx, vrefch4.tex.

- vcmdwin.cxx: This class shows a window with various command panes. The

144

"vCmdWindow" class is derived from the "vWindow" class. This class is intended as a class that serves as a main control window containing various "vPane" objects such as menu bars, canvases, and command bars.

- vcmdwin.cxx related components: vcmdwin.h, vpane.h, stdlib.h, vrefch5.tex.

- vrefch4.tex: This document covers the top level classes for building applications. The classes covered in this document include "vApp" which is the base class for applications, and "vAppWinInfo" which is a utility class used to interface views to models.

- vrefch4.tex related components: vapp.cxx, vawinfo.cxx.

- vrefch5.tex: This document covers the classes used to build windows and command windows. The classes covered in this document include "vCmdWindow" which is the class used to show a window with various command panes, "vCommandPane" which is used to define commands on a command bar, "vMenu" which is used to define pull down menus, "vPane" which is a base class for various window panes, "vStatus" which is used to define label fields on a status bar, and "vWindow" which is a class used to show a window on the display.

- vrefch5.tex related components: vcmdwin.cxx, vcmdwin.h, vapp.cxx, vcmdpane.cxx, vmenu.cxx, vstatusp.cxx, vwindow.cxx.

A reuser and a maintainer can also ask the librarian to purchase from external suppliers some reusable components that might be reused in order to maintain an existing system. The librarian can hold or reject their requests for new reusable components. Otherwise, he can decide to acquire the requested component. As shown in Figure 4.16 of Chapter 4, the reusable components purchased from an external supplier, are registered with TERRA's repository through a process of classification. The librarian/domain manager then needs to inform a reuser, a maintainer and a configuration manager that new reusable component versions are now available.

After the specification of the reusable component has been registered the file associated with the reusable component should also be stored into a reuse library controlled by RCS (Revision Control System).

Indexing using freeWAIS-sf-2.0's Field Option

"waisindex" of freeWAIS-sf-2.0 generates a number of index files by using the format of the files 'v.fmt' and 'v.fde' which have already been created manually. The command used for indexing[3] is of the form "waisindex -t fields -d v ../summary/sum_S0001.txt". "waisindex" generates or modifies the following files: v.src, v.fn, v.hl, v.doc, v.cat, v.dct, v.inv, v.stop. If the "-t fields" option is used to index registered files, a dictionary and an inverted file are created for each field of a format description file 'v.fde'.

7.4.2 Scenario # 2: Procedure of Change Control for an Existing System

This scenario presents an example where the implementation of the CR is completed through the process of maintenance since the reuser can not find out any possibly reusable components from the reuse library.

Issue and Analyse a Change Request, and Fill out the CR Form

Although the change request for an existing system can be issued by a maintainer, a reuser, or an end-user, this scenario supposes a maintainer, "Peter Johnson", who is in charge of maintaining the V system, has created a change request which is shown in Figure 7.1. A CR is identified by the next available number such as "CR001". The

---

[3]After running this command, the librarian should change the permissions of all database files within a directory "wsdb" to 660 in order to give a write permission for automatic indexing to an HTTP server.

initial status of a CR should be "Issued" and then changed into "Approved". The maintainer wanted a functionality to specify the size of a menuless and canvasless V application and also a functionality to support computations to be performed continuously. The type of this CR is "perfective maintenance" whereas the change request for "corrective maintenance" should be analysed in order to check whether it came from a user's incorrect operation of an existing system. In principle, an engineering change proposal (ECP) is created after analysing a CR, but this case study excludes an ECP as the CR includes some items of the ECP.

While the CR is analysed by a maintenance team the reuser can also initiate the first step of the reuse process, called "understand a CR". He recognised that after understanding the CR and searching for reusable components, there was no reusable component revision that might be used for 'white box' or 'black box' reuse in order to meet the change request from the maintainer.

Evaluate a Change Request and Fill in the Change Approval Form

The CRs accepted by a maintenance team are forwarded to the Configuration/Change Control Board (CCB). The CCB is responsible for the integrity of all software components, for evaluating/approving CRs and for monitoring the implementation of approved CRs. As shown in Figure 6.11 in Chapter 6, one of the CCB members fills out a change approval form by referencing the contents of the change request form that has already been filled in. The initial status of a change approval form should be "Issued" and then changed to "Approved", "Held" or "Rejected". After evaluating the contents of a change approval form, the CCB decided to approve the change request, changed the status of the change approval form from "Issued" to "Approved", and informed the maintainer and reuser that the CR had been approved by the CCB.

The maintainer needs to find out possible solutions to meet new requirements originating from the CR. As shown in the field "Description of Change" in Figure 6.11

147

Figure 7.1: A Change Request, CR001

in Chapter 6, the maintainer described briefly which part of an existing component should be changed. The change approval should be accompanied by a specification of redesign used to implement the CR. The maintainer needs to know what is involved in making the changes. Thus, he analysed the effects of the change on other components by using an impact analysis. He should cooperate with a configuration manager who is an expert on dependencies and relationships between components. Even though the maintainer is involved in an impact analysis, the CCB is in charge of performing an impact analysis to find out the components that a specific change effects. After completing the impact analysis, a project manager estimated resources including the implementation cost to complete the change request and change approval forms, "CR001 and CA001". The CCB needs to make clear that the change is cost effective. In other words, the implementation of the change request must carry benefits which outweigh the cost of implementing the CR. He also made a plan to implement the change using allocated resources.

Even though the reuse process may continue if the reuser can find potentially reusable components and the CR is approved by the CCB, the maintenance process will continue in the case of this CR. The subsequent steps of the maintenance process are as follows: maintenance implementation, revalidation (integration and regression testing), and re-insertion. In this case study, the details of the maintenance process after implementation are skipped although the refined activities of these processes have been included in the MwR (Maintenance with Reuse) model.

After successful integration and regression testing for revalidation, the project manager has allowed the maintainer to check the new version into the production library, and notified the reuser, librarian and configuration manager that the new component is now available. After the librarian and Quality Assurance manager reviewed the quality of the new version of the component they decided to register the version with a reuse library. The librarian stored the new version in the reuse repository using the RCS command. Thus, the component "vapp.cxx" has two revisions (i.e., R 1.1 and R 1.2) in both the reuse library and the production library.

Although the total numbers of revisions for these two libraries are different from each other, the same revisions should use the same version numbers in order to have good traceability.

7.4.3 Scenario # 3: Procedure of Combined Change Requests from a Reusable Component and an Existing Component

A few months later, two change requests for a reusable component and an existing component, 'vapp.cxx revision 1.2', have been issued by a librarian/domain manager and a maintainer, respectively. This scenario is an example of merging two CRs, and the implementation of the CRs follows the steps of the maintenance process as the reuser can not locate any reusable components which might be reused for 'white box' reuse and 'black box' reuse.

Issue and Analyse a Change Request, and Fill out the CR Form

Firstly, the change request for a reusable component, 'vapp.cxx revision 1.2', has been issued by a librarian/domain manager, "David Spencer". He wanted to upgrade the reusable component in order to support a different platform (i.e., Motif) by using "adaptive maintenance". Thus, he issued a change request(CR) to deal with this requirement. He filled out the CR form as shown in Figure 7.2, where the CR number was 'CR002', the component ID was 'S0001', the component name was 'a base class for building V applications', and the type of maintenance was "adaptive maintenance".

Some days later, "Ian Jones", who is one of the maintainers of the 'V' system, issued another change request, i.e., 'CR003', to provide an application 'vapp.cxx' with more functionality as shown in Figure 7.3. The related component ID, component name and type of maintenance in the CR form were 'S0001', 'a base class for

Figure 7.2: A Change Request, CR002

building V applications', and 'perfective maintenance', respectively.

Evaluate a Change Request and Fill in a Change Approval Form

Two CRs have been forwarded to the CCB which is responsible for approving the change request and monitoring implementation of the CR. Eddy Davis, who is a member of the CCB, displayed a fill-out form for change approval shown in Figure 6.11 in Chapter 6 by entering the CR number, i.e., 'CR002 or CR003', which is related to the fill-out form for change approval.

Using the fill-out form shown in Figure 6.11, the CCB entered corresponding change request numbers and the system's name, i.e., 'CR002 and CR003, and V', respectively. After an impact analysis, the CCB could find out which components the two CRs might affect. The CCB then filled in the field, related component ID and version as follows: 'vapp.cxx(S0001) R 1.2, vawinfo.cxx(S0002) R 1.1, vcmdwin.cxx(S0003) R 1.1, vrefch4.tex(R0001) R 1.2, vapp.h(S0004) R 1.2, vwindow.h(S0005) R 1.1, vcmdwin.h(S0006) R 1.2, vctlclrs.h(S0007) R 1.1, and vcolor.h(S0008) R 1.1'.

The reusable component, 'vapp.cxx', has been used to build systems such as 'V, X and Y'. The product lines that these systems belong to, are PL #1 for the 'V' system, and PL #3 for 'X' and 'Y' systems. After the CCB compared the cost with the benefits from the implementation of the CRs, the members of the CCB decided to approve the CRs. Additionally, the CCB decided to deal with the above two change requests in an identical change approval form as those CRs were associated with the same component. The librarian and Quality Assurance manager of the CCB wanted the requirements of these CRs to be included in the reusable component, 'vapp.cxx'. A project manager and a maintainer need to find out solutions to change the related component against the requirements of CRs and then to fill in the field of the specification of change. Finally, the due date of change and estimated resources (i.e., manpower and cost) should be decided and entered.
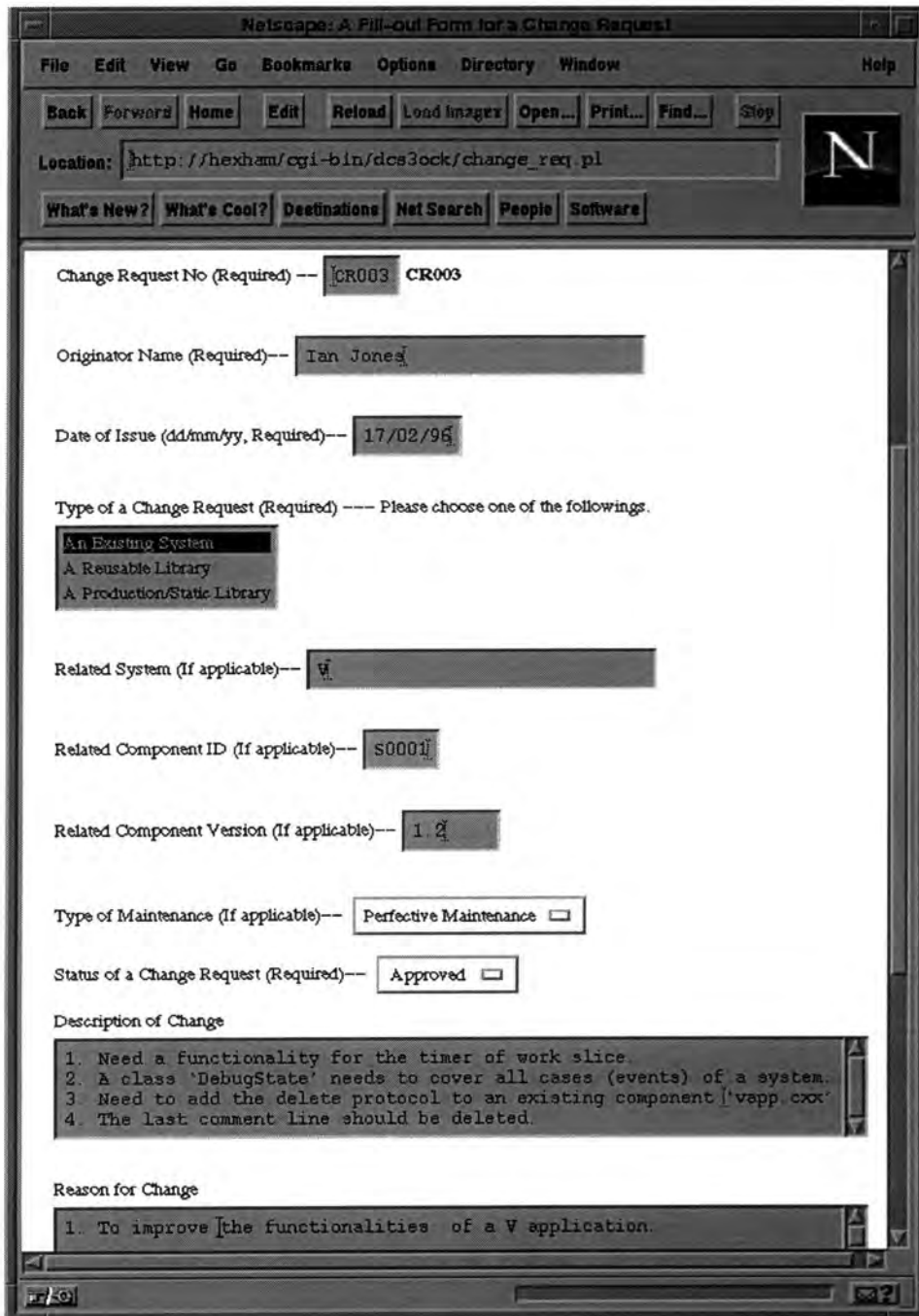
Figure 7.3: A Change Request, CR003

The estimated manpower depends upon the deadline when the implementation of change requests should be completed by. In other words, an urgent change request requires more man-months in order to finish implementing as soon as possible. After the CCB submitted the fill-out form for change approval it received a confirmation message: "The change approval form, CA003, has been successfully inserted as a file name called CA003.txt".

While the CCB analysed two CRs (i.e., CR002 and CR003) the reuser needed to search the reuse library to find a potential reusable component. He realised that there was no component to be reused to maintain the reuse library and legacy system. After the maintainer was told about this and the CCB approved the CRs, he started implementing the CRs under the supervision of a project manager. The CCB had already decided to merge these two CRs into one change approval, so the maintainer could check out any of the two components. Using the results from an impact analysis, the maintainer finished modifying the appropriate components, followed by revalidation, i.e., integration testing and regression testing.

After the integration testing and regression testing had been performed successfully, the configuration manager allowed the maintainer to check the new revision into the production library and notified a librarian that the change of a component associated with a reusable component had been made. A project manager asked the maintainer to release the new revision to end-users. A new revision should not be released until it meets the quality requirements of Company XYZ. The librarian/domain manager also checked the new revision into the reuse library as the librarian and Quality Assurance manager had already agreed to register the reusable component version with the reuse repository. He then sent reusers and maintainers the message indicating that the new revision of the reusable component is now available from the reuse library. The new revision of the component, 'vapp.cxx', was numbered as R 1.3.

7.4.4 Scenario # 4: Procedure of Change Control for an Existing System

This scenario shows an example where the implementation of the change request for a legacy system can be performed using reusable components as the reuser can find out a potentially reusable component from the reuse library. Thus, the implementation of the CR can be supported by a process of reuse.

Issue and Analyse a Change Request, and Fill out the CR Form

The maintainer, "Steve Smith", created a change request since the users of the V system had reported that there are inconsistencies in the order of width and height parameters in previous versions of V. After completing the CR, he passed a copy of the CR onto the reuser, "Rick Smith", to let him search for potential reusable component versions. "Steve Smith" filled out the CR form whose CR number is 'CR004', as shown in Figure 7.4. After analysing the CR, the maintenance team decided to accept the CR and to forward it to the CCB. Thus, "Steve Smith" changed the status of the CR from 'Issued' to 'Approved'.

Searching for Reusable Components

The reuser, "Rick Smith", reviewed the change request (CR) which has been passed from the maintenance team in order to understand the requirements of reuse included within the CR. After identifying keywords or names of components, he clicked on 'Reusable Components Search' of the TERRA home page and then also clicked on 'Search by Keywords'. Using a search command with a field option such as "ci=s0001", "Rick Smith" obtained the results of a search that output a headline that consisted of component identifier, component name, author name, operating systems, computer language, and component format. As shown in Figures 7.5 and 7.6, he then retrieved a specification of the reusable component, 'S0001', in order

Figure 7.4: A Change Request, CR004

Figure 7.5: Specification of the Reusable Component: S0001 Part #1

to see the details of the reusable component as shown in Figure 7.6. After clicking on 'View the History of Change' shown in Figure 7.6, "Rick Smith" reviewed the revisions of the component to decide if the requirements of the CR can be met by any reusable component version within the reuse library.

Evaluate and Adapt Reusable Components

After reviewing the history of the change, the reuser compared several revisions using the difference list between different versions. The reuser, "Rick Smith", realised that the reusable component revision 1.4 satisfied the requirements of the CR in part, so he planned to adopt 'white box' reuse in order to implement the CR if the CCB approved the CR. In other words, this revision had already included the CR requirements 1 and 2, but it had not included the requirements 3, 4 and 5. In the mean time, the CCB started reviewing the CR to make a decision whether or not
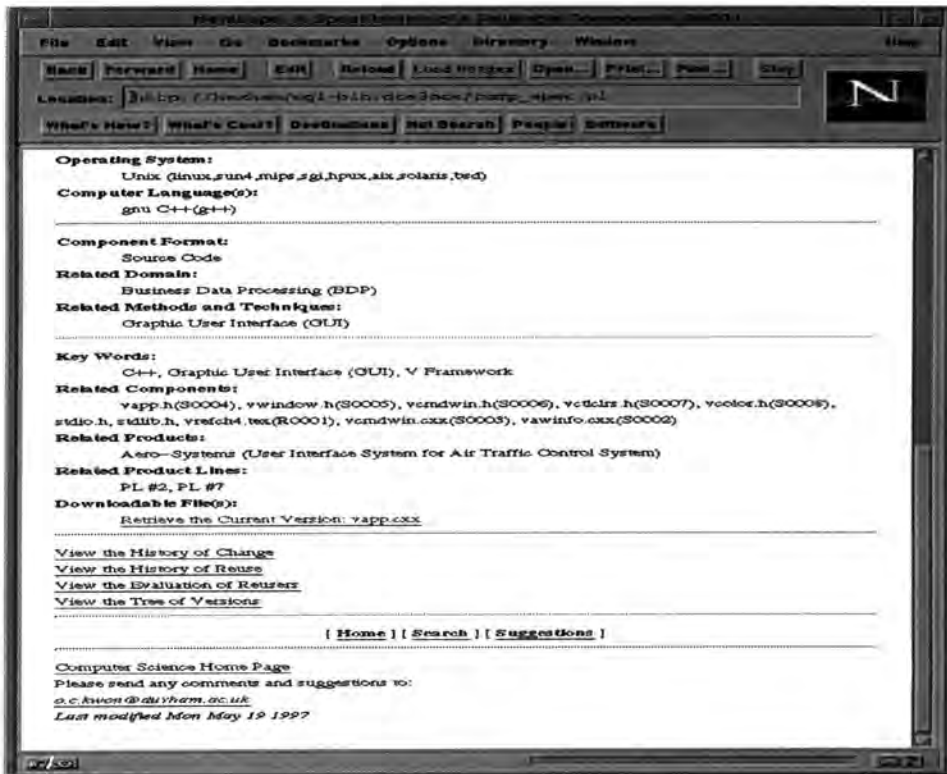
Figure 7.6: Specification of the Reusable Component: S0001 Part #2

it should approve, reject, or hold the CR.

Evaluate/Approve a Change Request and Fill in a Change Approval Form

After the CCB received and evaluated the CR, it decided to approve the CR in part since some requirements (i.e., requirements 3,4,5) are only associated with another component. Thus, requirements 1 and 2 of the CR have only been approved by the CCB whereas requirements 3, 4 and 5 have been held, waiting for review later. The CCB notified the reuser that the CR had been approved. In addition, the CCB informed the originator, "Steve Smith", that some change requirements had been held to be dealt with later. When a member of the CCB filled out an entry form for change approval he omitted the field 'Specification of Change' in Figure 7.8 but filled out all the fields in Figure 7.7. The reason why he can omit the field is that the reuser can reuse the reusable component to implement the CR. The CCB assigned
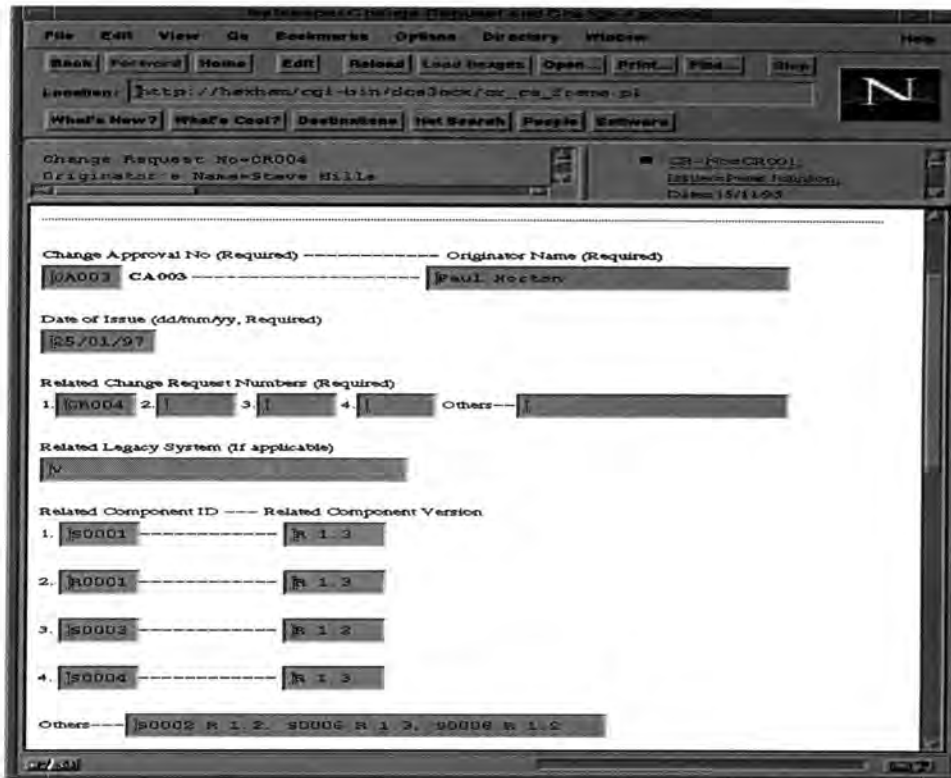
Figure 7.7: A Fill-out Form for Approval of the Change Request CR004: Part #1

lower resources to the estimated manpower and cost because the reuser notified the CCB that 'black box' reuse was applied to implement the change request, 'CR004'.

Integration of a reusable component into an existing system

The reuser, "Rick Smith", planned to adopt 'white box' reuse at the beginning of reviewing the CR, but he could apply 'black box' reuse to implement the CR since the requirements(i.e., reqs. 3, 4, 5) of the CR related to 'white box' reuse, had been held for the moment. Thus, the reusable component 'S0001: vapp.cxx' revision 1.4 can be perfectly applied to meet the requirements of the CR, 'CR004'. 'Black box' reuse does not require any regression testing. After the tester(QA person) finished successful integration testing, the reuser informed a maintainer and configuration manager that the CR, 'CR004', had been completed. The configuration manager allowed the maintainer to check the new revision in a production library. The project

Figure 7.8: A Fill-out Form for Approval of the Change Request CR004: Part #2

manager then asked the maintainer to release the new version, 'vapp.cxx' revision
1.4, to end-users.

Updating of Reuse History

The reuse report might bring benefits to other reusers since the information on
reuser's experience can give confidence to reusers who are reluctant to reuse a
reusable component. As shown in Figures 7.9 and 7.10, the reuse report can be
identified by the component ID and version.

If the reuser adopts 'black box' reuse, he does not need to fill in the fields 'summary
of modifications' and 'how much you have modified'. The fields 'related product
lines' and 'related products' show the family and system, respectively, for which
the reusable component has been reused. Using the field 'other comments', the
reuser also filled out reuser's experience and benefits obtained from the use of the
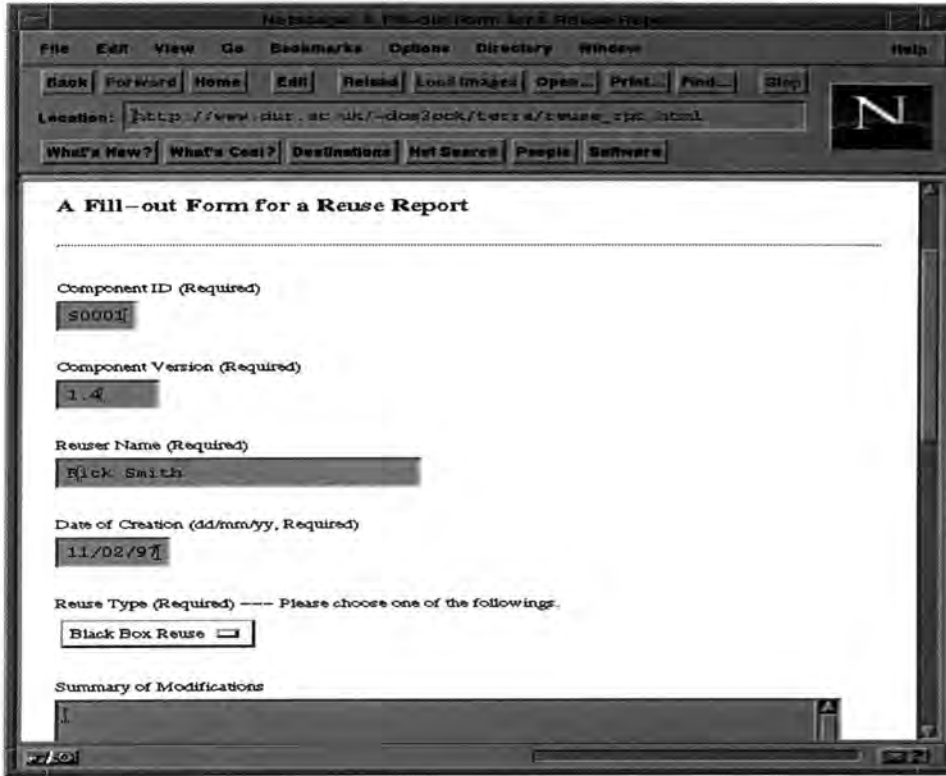
Figure 7.9: A Reuse Report Part #1 for the Reusable Component, S0001: vapp.cxx

corresponding reusable component.

7.5 Review of the Case Study

This chapter has shown the case study based on the sample data 'V', which is a "Freeware Portable C++ GUI Framework for Windows and X". This case study has used four scenarios which are convenient for the purpose of this discussion, to demonstrate the use of a reuse server named TERRA that has been implemented using the MwR model. An evaluation will be carried out in the next chapter.

The first scenario shows registering reusable components with a reuse library. A librarian/domain manager is in charge of evaluating, classifying, storing, changing and notifying reusable components. He needs to cooperate with a Quality Assurance manager when evaluating potential reusable components. The MwR model supposes

Figure 7.10: A Reuse Report Part #2 for the Reusable Component, S0001: vapp.cxx

that the candidate of reusable components can come from the reuser, maintainer, or external supplier. In other words, the reuser can submit potential reusable components to the librarian/domain manager if he thinks reusable component versions created as a result of 'white box' reuse could be a candidate for a particular reusable component. The maintainer might extract some potential reusable components from an existing system. The reuser or maintainer can also ask a librarian/domain manager to purchase reusable components from an external supplier.

The second scenario has been used to show how the model of the change request (CR) for a legacy system works. The change request created by the maintainer, that is originally issued by an end-user, can be implemented by the steps of a reuse process or maintenance process. In this case, the CR has been implemented by a maintenance process since the reuser came to know that there was no reusable component that could be used in 'white box' or 'black box' reuse. Thus, the reuser needs to closely cooperate with the maintainer in order to decide if he can continue

to follow the process of reuse.

The third scenario has been introduced in order to present the procedure of change control for a reusable component. Although this scenario is associated with two change requests which have been issued by a librarian and a maintainer, its aim is to show an example of the CR for a reusable component, where there is no reusable component version to meet the requirements of this CR. Thus, the procedure of change control has followed the process of usual maintenance. However, as the reusable component has been reused and modified in order to implement the CRs for an existing system and a reuse library, this scenario can be classified as 'white box' reuse.

The fourth scenario has been chosen to give an example of a CR that has been issued by a maintainer and can be implemented by a reusable component version within a reuse library. This scenario presents a process of reuse that can be fully supported by 'black box' reuse and meet the change request created by a maintainer in order to implement changes to an existing system.

7.6   Discussion of the Case Study

The objective of this case study is to review the 'Maintenance with Reuse (MwR)' model and TERRA tool that support a legacy system and a reuse library through a scenario based approach. There exists a very strong relationship between software maintenance and software reuse in terms of Software Configuration Management (SCM). SCM enables a reuser and maintainer to solve some problems with reuse and maintenance. In particular, software components within an existing system and a reuse library are subject to evolution over time, so software maintenance and reuse require SCM. The processes of software maintenance and reuse also have similar activities that enable us to construct an integrated model. For instance, the analysis of a CR within a maintenance environment is associated with the subactivities of

163

both the maintenance process and reuse process. In addition, both the maintenance process and reuse process have the same sub-activities such as "integration" and "re-insertion".

In this research, the author developed an integrated model of the reuse process and the maintenance process within an SCM environment. The MwR model consists of four major activities: configuration management (CM) process, reuse process, maintenance process and administration of a reuse library. The activities of configuration management support auxiliary(subsidiary) functionalities that enable the integration of a reuse process with a maintenance process within a software maintenance environment. The CM process can also manage the evolution of a reuse library. The processes of reuse and maintenance in addition to the use of a reuse library should be supported by SCM in order to manage changes to components which exist within these processes.

A Configuration Management (CM) process has been used for integrating the maintenance process and reuse process within a maintenance environment. The CM process has enabled the reuser and maintainer to control the evolution of the reuse library and legacy system. In addition, the functionalities of CM have provided the reuser and maintainer with a variety of reports in order to enhance the visibility and traceability which might be useful for implementing the CR and managing the reuse library. The TERRA prototype has been developed on the WWW to support the four major processes of the MwR (Maintenance with Reuse) model.

Chapter 8

Evaluation of the MwR Model and TERRA Prototype

Section 8.1 describes some amendments to the model that have been made as a result of the case study presented in Chapter 7. Section 8.2 addresses why the MwR model and TERRA can make systematic reuse effective and how the model and tool can be customised for different organisations. Section 8.3 presents some benefits that they can bring to the software maintenance and reuse community, followed by several shortcomings to be addressed.

8.1 Modification of the MwR model and TERRA

In order to enhance the applicability of TERRA within a real organisational environment, the author has modified the four major processes that make up the MwR (Maintenance with Reuse) model using some feedback obtained during the case study. The maintenance process and reuse process should be a cooperative procedure as shown in the MwR model since there exists a strong relationship between the two processes. The continuation of activities of the reuse process depends on whether reusable components exist within a reuse library or if the CCB approves

the CR.

The functionalities of a librarian/domain manager consist of "populating the library", "change control of reusable components", and "notifying changes in the library". The sub-activity, "notifying changes in the library", has been added for the administration of a reuse library in order to inform reusers and maintainers of the evolution and availability of reusable components.

The author has separated a model for management of a reuse library from an integrated model of reuse and maintenance. This research has focused on two kinds of procedures of Software Configuration Management (SCM): one is a procedure of change and version control for a maintenance environment that is supported by a reuse library, and the other is a procedure of change and version control for a reuse library. These two procedures are shown in Figures 4.14 and 4.16 in Chapter 4, respectively. The configuration manager is in charge of the management of evolution of an existing system. On the contrary, the domain manager has responsibility for managing the evolution of a reuse library. Thus, the separation of these two distinct activities has reduced the complexity of the MwR model and also enabled the author to successfully complete a scenario based case study.

As shown in Figure 4.14, the integrated diagram of the reuse process and maintenance process includes the diagram associated with the functionality of a configuration manager. After change requests (CRs) are passed onto the reuser, the maintainer and the configuration manager, the configuration manager who is one of the CCB members needs to monitor the process of implementation of CRs.

Since the process of maintenance includes the sub-activity of "re-insertion", the process of reuse also includes the sub-activity "re-insertion", which had been initially contained in the sub-activity "integration". Having the same sub-activities of "re-insertion" in the reuse and maintenance process makes it easy to maintain the integrated process of reuse and maintenance. One of the sub-activities of the maintenance process, "revalidation", has been changed to "integration" since it is

exactly the same as that of the reuse process and the two processes need to maintain consistency with each other.

8.2 Introduction of the MwR Model and TERRA to an Organisation

Through the four scenarios of the case study, we have seen how the change requests (CRs) have been implemented using an approach that integrates the maintenance process and reuse process. After conducting the case study, the author has realised that the MwR model and its prototype enable the introduction of systematic reuse to a software maintenance environment, because the MwR model and TERRA have considered the aspects of the reuse process and organisation that are key factors for successful reuse. In addition, the process of reuse has been set up within a maintenance environment, and the roles of a librarian/domain manager, a reuser, a maintainer, a configuration manager and a Quality Assurance manager have been identified in terms of the structure of an organisation.

The MwR model and the TERRA prototype should be customised when they are introduced into various organisations. Depending on the size of an organisation there may or may not be a reuser. In a large organisation, there may be a reuser as well as a maintainer, as described in the MwR model. In a small organisation, the duties of a reuser can be taken on by a maintainer. In addition, when the reuse technology is first introduced into the organisation, both a reuser and a maintainer are necessary in order to standardise the activities of the reuse process and to encourage a maintainer who is not willing to reuse, to use reusable components in the maintenance of an existing system. At the beginning of the introduction of reuse, a maintainer is usually not willing to use reusable components because of 'NIH (Not Invented Here)' syndrome.

An organisation requires the appropriate level of change control. If change control

is too restrictive then the process of maintenance might be interrupted and will last longer. If change control is too weak then software components may lose their visibility and traceability, so the quality of an existing system and a reuse library will be reduced greatly. The major factors that affect the appropriate level of change control are the size of the project team and risks related to the change. A large project team needs more formal procedures than a small project team because a larger project is associated with a lot of components and should be divided into several subsystems which require their own CCBs. The change control procedure for safety critical software must be strict and formal as incorrect changes concerned with safety critical systems may result in the loss of human property and life.

The CCB has other synonyms such as review board, change control board or quality control. In a small organisation, the role of the CCB can be taken on by a single person, for example, a project manager. Although the size of the CCB may be varied it should always comprise a configuration manager, a Quality Assurance manager and an end user's representative.

Although there are two approaches to Software Configuration Management (SCM), i.e., a tool based approach and a paper based approach, this research has implicitly assumed a tool based approach as SCM without tool support might incur many burdens on users. Although TERRA has been initially developed for use by an Intranet, it can also be run on the Internet by the introduction of a user-id and password. Thus, TERRA can be effectively introduced into an organisation that has several geographically separate branches or offices.

In order to be used in an Intranet across an organisation, the TERRA tool needs to keep version numbering between the two repositories (i.e., a reuse library and a production library for a legacy system) consistent. In addition, the two identical components which are checked into the two repositories, should be numbered with the same version number so that a configuration manager and domain manager/librarian can control the evolution of every component efficiently.

As shown in Figure 3.1, when the reuser adopts 'black box' or 'gray box' reuse the system of version numbering follows that of the reusable component such as R 1.1, R 1.2, R 1.3, etc. On the contrary, the system of version numbering of 'white box' reuse uses the same version numbering system as variation so that multiple copies of the reusable component can be managed and kept consistent with version numbers of 'black box' and 'gray box' reuse. However, the version number of 'white box' reuse is identified as R 1.1.1.1, R 1.1.1.2, R 1.1.1.3, etc, whereas that of variation is identified as V 2.1.1.1, V 2.1.1.2, V 2.1.1.3, etc. For instance, in Figure 3.1, the reuser A used a specific reusable component revision 1.2 (R 1.2) from the reusable component database without any modification (i.e., 'black box' reuse), so he needed to use the same number as the reusable component. The reuser B used a reusable component revision 2.1 (R 2.1) after modification (i.e., 'white box' reuse), and identified the version number of the component as R 2.1.1.1. The maintainer A retrieved and used a reusable component variation 1.2.1.1 (V 1.2.1.1), so he used just the same number as the version number of the reusable component.

8.3 Criticism of the MwR Model and TERRA: Benefits and Limitations

Since there exist several similarities between reuse and SCM and between reuse and maintenance, an integrated approach of reuse, maintenance and SCM has been investigated. Moreover, many tasks which should be carried out in order to reuse software components for software maintenance, can be solved using SCM. The TERRA tool has been implemented in order to provide automated support for Software Configuration Management (SCM) of a reuse library and a legacy system.

When performing a case study, the author has used several examples of the CR to explain and evaluate the TERRA prototype and MwR model. As a result of the case study, many strengths have been observed. The TERRA prototype is an

easy-to-use, ease-to-learn, platform independent, and very flexible reuse server. All these strengths have resulted from the fact that TERRA has been implemented on the WWW. TERRA has flexibility as it can be used in both an Intranet and the Internet.

Through the case study, the author has also found out that TERRA has some weaknesses, as described below:

- TERRA does not provide formatted reports on 'the history of reuse', 'change request (CR)' and 'change approval (CA)' although it produces a formatted report for the description of a reusable component that is one of more important reports to be used by reusers and maintainers.

- TERRA does not allow users to generate all reports related to the process of Software Configuration Management (SCM) and administration of the reuse library. For instance, if TERRA can produce all reports associated with implementation of a specific change request (CR), including 'the evaluation of reusers' and 'the tree of version', the traceability and visibility of change control will be greatly improved.

- The functionalities for 'the tree of version' and 'the evaluation of reusers' have not been implemented although these functionalities are included in the sub-activity "status accounting" of the MwR model. 'The tree of version' shows the evolution of a component graphically. 'The evaluation of reusers' provides information which is valuable for the reuse metrics.

- TERRA does not provide fill-out forms for updating of input data after the fill-out forms have been stored in a reuse library. Although the users can update the related text files directly and experienced users prefer direct updating of the files, the fill-out form for updating can provide users with a friendly entry form that might be very helpful for most novices.

- TERRA does not include the fill-out form for evaluation of a reusable compo-

nent which is registered with a reuse library. The evaluation of the reusable component is performed by a QA manager, a reuser or a maintainer.

The reports and functionalities described above seem to us useful, but time constraints made it impossible to implement these in the TERRA prototype. The case study has used four scenarios which can be applied to the implementation of various change requirements created from a maintenance environment and a reuse library in order to evaluate the MwR model and the TERRA prototype. Although there exist some shortcomings in TERRA such as lack of all reports on SCM, it is believed that the TERRA prototype tool will become the backbone for systematic reuse, thereby bringing a lot of benefits to an organisation, in particular, the maintenance community and reuse community.

In conclusion, this research, which integrates reuse with maintenance within an SCM environment, presents one example of tool integration, thereby allowing users to gain remarkable improvements in both software quality and productivity. In addition it helps an organisation to enhance the effectiveness of investment in the research of software engineering through simultaneous investigation into common activities of reuse and maintenance. This research will enable the software reuse process to work as an integral part of software engineering and likewise software reuse and SCM to be introduced into an organisation which maintains software systems.

Chapter 9

Conclusions

In order to solve effectively many problems with software reuse and maintenance, these problems need to be tackled together, through an integrated approach of these two research fields. This research has therefore developed an integrated process model for 'Maintenance with Reuse (MwR)', that supports Software Configuration Management (SCM) for a reuse library which is actively maintained for use in a software maintenance environment. The prototype tool of the MwR model, TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library), has been implemented on the WWW so that the prototype can provide portability, traceability, integration with existing tools, and a distributed maintenance environment. The TERRA prototype has been tested and evaluated through a scenario based case study that has used 4 different types of scenario.

The integrated model of MwR and its prototype can overcome many problems that exist in software maintenance and reuse through introduction of SCM functionalities into a maintenance environment, thereby enhancing productivity and quality of software, and also reducing the costs of implementing changes to existing systems. This chapter reviews the results of the work that has been carried out through this research, and evaluates generally the MwR model and the TERRA prototype, followed by directions for further research that can be performed based on this

172

research.

9.1   Results of this Research

Models and prototypes that support the process of 'Maintenance with Reuse (MwR)'
have been developed. The major results of this research as described in the criteria
for success in Chapter 1 are as follows:

- *To develop models for 'Maintenance with Reuse (MwR)' that supports the evo-
  lution of a legacy system by a reuse library.* The models can support most
  activities related to the reuse, maintenance, and configuration management
  (CM) processes. The method of MwR integrates a reuse process with a main-
  tenance process through the introduction of Software Configuration Manage-
  ment (SCM) into these two processes, thereby solving many SCM related prob-
  lems within software maintenance and reuse. This integrated model includes
  cooperative activities between the reuser and the maintainer, and functions of
  change and version control. Change control procedures to support the evolu-
  tion of an existing system through a reuse library have been developed. These
  procedures show how the change requests (CRs) initiated with respect to a
  legacy system are analysed, approved, implemented, tested and released.

- *To develop the procedure of change and version control for reusable components
  within a reuse library.* Although the reuse library only supports the process
  of 'black box' reuse, the reusable components within the reuse library are
  subject to change over time. Thus, all change requests should be controlled,
  managed and propagated through a strict change control procedure as in a
  legacy system. Additionally, if the reuse library can store multiple versions
  of a reusable component through the processes of version control and quality
  control, the availability of the reuse library will be increased greatly.

- *To model and implement administrative functionalities associated with a reusable software library.* The functionalities modelled include classification, registration and retrieval of reusable components, and notification of changes to reusable components.

- *To produce information (i.e., status accounting) related to reuse and SCM.* Some reports that contain information on reuse and SCM, have been developed. For instance, the report of change history provides reusers and maintainers with information on who made the changes, what changes have been made, when the changes were made, and why the changes were made. The report of reuse history includes some information on the reuser's experience with reusable components, such as the type of reuse, summary of modification, relevant product, product line and domain, and comments that might be useful for potential reusers.

- *To develop a prototype that supports the 'Maintenance with Reuse (MwR)' model.* The prototype of a reuse server called TERRA (Tool for Evolution of a Reusable and Reconfigurable Assets Library) has been developed on the WWW. TERRA provides maintainers, reusers, librarians/domain managers, configuration managers, project managers, and quality managers with an automated tool that supports a maintenance process, a reuse process and an SCM process, together with administrative functionalities for a reuse library.

9.2 Assessment of this Research

This research has focussed on developing change control procedures for both a reuse library and an existing system, including most of the activities for these two processes within a maintenance environment. In particular, in this research the main functionality of SCM for the existing system is to establish procedures of the reuse process for implementing change requests (CRs) to a legacy system. The reuse process must be integrated into a software maintenance environment so that reusers

and maintainers can perform systematic reuse and enhance the effectiveness of reuse. SCM can be adopted as a minimum requirement for a maintenance environment because it provides the procedures to control and manage the evolution of a legacy system.

As shown in Section 2.4 of Chapter 2, there exist similarities and relationships between reuse, SCM and software maintenance. For these reasons, the MwR model has been built to combine the reuse process with the maintenance process through introducing SCM functionalities into a maintenance environment. Since reuse libraries and servers currently available do not have the flexibility to be integrated into users' development and maintenance environments, they have not been successful so far.

TERRA is an automated framework that supports the processes of reuse, maintenance and SCM, including the evolution of the reuse library. Although some reuse servers are now available on the WWW, most of them do not completely support the functions required to control the evolution of reusable components in a reuse repository, and the changes of legacy systems within a maintenance environment. In particular, there are no reuse servers which support efficient version control within a maintenance environment. As the TERRA tool has been developed on the WWW, it can provide good capabilities of usability, portability, traceability, integration with existing tools, and construction of a distributed maintenance environment.

In order to make a reuse programme effective and viable, organisational problems as well as technological problems need to be solved. For instance, most models of reuse libraries do not consider the structural aspects of an organisation that might be needed for its application. However, TERRA's model identifies the roles of a *librarian/domain manager, a configuration(component) manager, a reuser and a maintainer.* These aspects help an organisation successfully introduce the server into its maintenance environment.

The MwR model and its prototype enable the introduction of systematic reuse into a software maintenance environment because they have considered the aspects

of the reuse process and organisation that are key factors for successful reuse. This research also helps an organisation to establish a software maintenance support environment because reuse and Software Configuration Management (SCM) are usually considered the core parts of a maintenance environment. In particular, this research has proposed a framework of a software maintenance support environment that includes a production library (repository) controlled and managed by SCM. It is expected that the MwR model and TERRA can be used as the keystone which constructs a software maintenance support environment.

The introduction of SCM functionalities into the reuse process and maintenance process, expedites the improvement of a maintenance process through standardisation that results in the success of systematic reuse. As SCM allows an organisation to standardise its development/maintenance process using the functionalities of SCM, the integrated model described here enables an organisation to introduce a process assessment and improvement programme through the establishment of the level 2 (repeatable) of SEI's Capability Maturity Model (CMM). A Reuse Capability Model (RCM) should be used as a guide to selecting improvement strategies by measuring current reuse capabilities and identifying the issues most critical to reuse improvement. As most of the organisations are still placed on the *'initial'* level of the CMM model, they require their processes to be moved towards a higher level.

The TERRA tool provides various reports for the reuser, maintainer, librarian, domain manager, configuration manager, project manager, etc. so that they can monitor and manage the process of evolution of a legacy system and a reuse library using the functionalities of visibility and traceability from the reports. These reports include a specification of a reusable component, a change request form, a change approval form, a change history report, a reuse history report, and a difference list of versions.

The evaluation of the MwR model and TERRA prototype has been made through the scenario based case study. The case study has used 4 different types of scenario in order to evaluate how the MwR model and TERRA deal with evolution of both a

176

reuse library and an existing system through change and version control procedures for changes made to software components, and how the reuse library is maintained. The results obtained from the evaluation have been used for improvement of and amendments to the MwR model and TERRA prototype.

Due to time limitations, all activities of the reuse and SCM processes have not been implemented although most of the activities have been included in the MwR model. However, the model and TERRA prototype will bring many benefits to organisations when being used on an Intranet as well as the Internet, thereby leading to efficiency of investment in software engineering. If the common activities of reuse, maintenance and SCM are attacked at the same time, many problems within these three fields can be solved together, resulting in synergy. Even though return on investment will usually accrue in the long term, this integrated approach enables an organisation to advance the break-even point.

In conclusion, even though there exist some drawbacks in the MwR model and TERRA, it is believed that through the integrated approach of the reuse and maintenance processes within an SCM environment, this research will play a major role in building the backbone of a software maintenance environment that can overcome many problems with software reuse and maintenance, thereby resulting in significant enhancements of the productivity and quality of a software product including reduction of maintenance costs.

9.3   Further Work

In order for the work described here to be feasible and effective, some further research needs to be done.

Firstly, both software reuse and maintenance as well as SCM require system modelling languages for software building. It is difficult to build complex systems from software components. To solve this problem, a better Syntactic Interconnection

Language such as SySL and PCL could be used for system modelling and version selection. Alternatively, a Component Description Language (e.g., OBJs, LIL and RESOLVE) that can support Semantic Interconnection Modelling which specifies the behaviour of a system as well as the structure of a system, could be enhanced and adopted.

Secondly, different data models, classification schemes and terminology of reuse libraries result in the inability to share reusable components between reuse libraries. Research into reuse library interoperability has been carried out and applications of the effective interoperation have already been made between some reuse libraries, but they have not been implemented completely. The TERRA reuse library could interoperate with other reuse libraries on the WWW to enable reusers to share reusable assets without direct access to other libraries.

Thirdly, TERRA has only adopted keyword searching by free text or fields as a representation method of reusable components. This is because Frakes' empirical study [41] shows that there is no significant difference in search effectiveness between the representation methods, and no method achieved a best or even adequate rating. However, the TERRA prototype could be redesigned to support multiple representation methods such as *attribute-value classification, enumerated classification, faceted classification* since more representation methods can increase the probability that relevant reusable components will be retrieved, and different users may have different preferences. In addition, in order to enhance the understanding of retrieved reusable components, a tool for domain analysis could be incorporated into the prototype.

Fourthly, this research has focussed on the process of 'Maintenance with Reuse (MwR)' that *maintains a legacy system through the reuse and SCM activities*, in contrast with the process of 'Development with Reuse (DwR)' that *develops a new system using reusable software*. Thus, in order for the above two processes to be successful, both MwR and DwR processes require the introduction of 'component engineering' that *populates reusable components* by using the 'Development for Reuse

(DfR)' process that *creates reusable specifications and code,* and the 'reverse engineering' process that *extracts reusable assets from existing systems.* Therefore, the MwR model and TERRA tool should be linked with the methods and tools associated with 'DfR' and 'reverse engineering'.

Finally, a good starting point for undertaking reuse is to assess the maturity level of organisational reuse with respect to the Reuse Capability Model (RCM) [32]. The result of the assessment will be used for establishing the reuse programme applicable to each organisation. Thus the Reuse Capability Model must be investigated in order to improve an organisation's reuse capability. A set of RCM guidelines to introduce SCM and software reuse into organisations based on the SEI's CMM needs to be developed.

Chapter 10

Publications and Reports

The author has published and written the following papers and reports during the course of this research.

1. O. C. Kwon, C. Boldyreff and M. Munro, "Scenario Based Case Study: A Maintenance with Reuse (MwR) Model and its Implementation", To be Submitted to *IEEE-TSE Special Issue on Scenario Management*, IEEE Transactions on Software Engineering, USA, January 1998.

2. O. C. Kwon, C. Boldyreff and M. Munro, "Software Configuration Management for a Reusable Software Library within a Software Maintenance Environment", Accepted for *the International Journal on Software Engineering and Knowledge Engineering (IJSEKE)*, Knowledge Systems Institute (KSI), USA, December 1997.

3. O. C. Kwon, C. Boldyreff and M. Munro, "An Integrated Process Model of Software Configuration Management for Reusable Components", In Proceedings of *the Ninth International Conference on Software Engineering and Knowledge Engineering (SEKE'97)*, June 18-20, 1997, Madrid, SPAIN. Also as Technical Report 11/96, Centre for Software Maintenance, Department of Computer Science, University of Durham.

4. O. C. Kwon and C. Boldyreff, "Software Technology Analysis Task for a Software Maintenance Support Environment", DiCE Project Report Sponsored by BT, in Collaboration with UMIST and Keele University, Centre for Software Maintenance, University of Durham, 1 April 1997.

5. O. C. Kwon, C. Boldyreff and M. Munro, "Integration of a Reuse Process and a Maintenance Process within a Software Configuration Management(SCM) Environment", In *Proceedings of the 8th Annual Workshop on Software Reuse (WISR8)*, Ohio State University, Columbus, Ohio, USA, 23-26 March 1997.

6. O. C. Kwon, "SEKE'97 Report: the Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE'97)", June 17-20, 1997, Husa Princesa Hotel, Madrid, SPAIN, In Will Tracz, editor, *ACM SIGSOFT, Software Engineering Note (SEN)*, Volume 22, Number 5, ACM Press, October 1997.

7. O. C. Kwon, "A Software Factory Based on Software Reuse", In Proceedings of *Software Centre 2005 Workshop*, Centre for Software Maintenance, Department of Computer Science, University of Durham, 15th December 1995.

8. O. C. Kwon and C. Boldyreff, "WISR8 Trip Report: 8th Annual Workshop on Software Reuse (WISR8)", 23-26 March 1997, Ohio, USA, In E. Ng, editor, *The BCS Reuse Specialist Group's Newsletter, Re-Print*, Issue 34 , October 1997.

9. O. C. Kwon and A. Jones, "An Application of the HAZOPS Technique to a Software Project Management Model", Technical Report, University of Teesside, January 1995.

Appendix A

Other Fill-out Forms and Tools
Used for this Research

A.1 More Fill-out Forms and Reports Produced by TERRA

Figures A.1, A.2, A.3, A.4 and A.5 are forms and reports that have not been included in Chapter 6 and Chapter 7.

A.2 freeWAIS-sf-2.0.65

A.2.1 The Format Definition File 'v.fmt' Used for Indexing

```
record-sep= /\f/    # formfeed (alias for \L)


layout=
  headline= /^ci=/ /^cn=/ 5  /^ci=*/
```
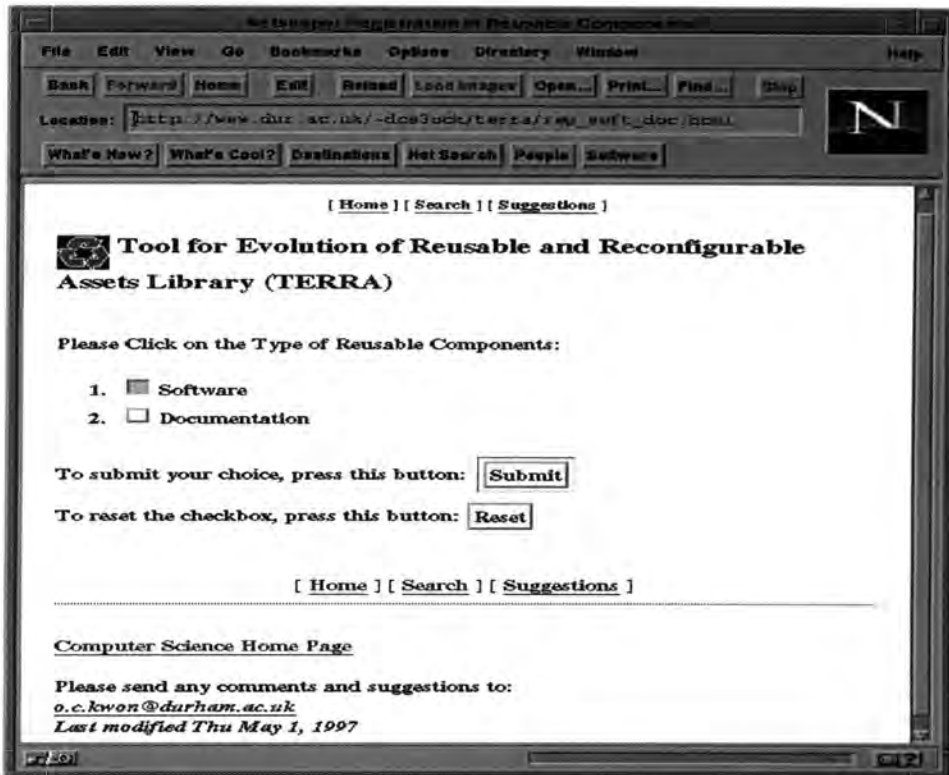
Figure A.1: Main Menu for Reusable Components Registration

```
headline= /^cn=/ /^an=/ 30 /^cn=*/

headline= /^an=/ /^dc=/ 15 /^an=*/

headline= /^os=/ /^cl=/ 12 /^os=*/

headline= /^cl=/ /^cf=/ 12 /^cl=*/

headline= /^cf=/ /^dm=/ 15 /^cf=*/

end=


region= /^ci=/ /^ci=*/
ci "Component-ID" TEXT BOTH
end= /^cn=/


region= /^cn=/ /^cn=*/
cn "Component-Name" TEXT BOTH
end= /^an=/
```
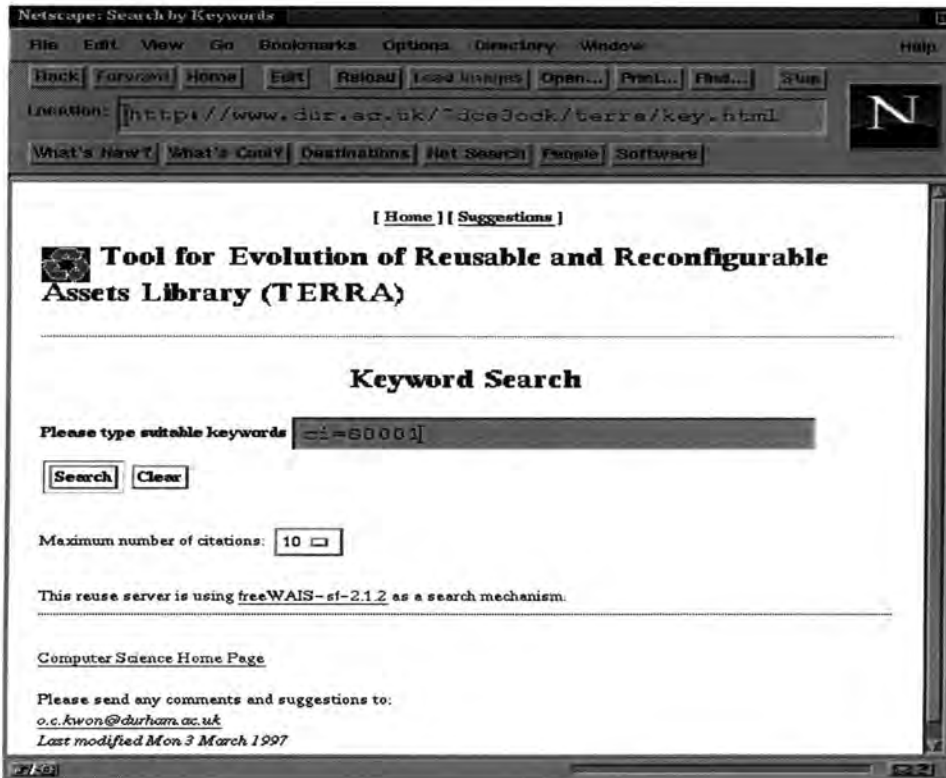
Figure A.2: A Fill-out Form for Search

```
region= /^an=/ /^an=*/

an "Author-Name" SOUNDEX LOCAL TEXT BOTH

end= /^dc=/


region= /^os=/ /^os=*/

os "Operating-Systems" TEXT BOTH

end= /^cl=/


region= /^cl=/ /^cl=*/

cl "Language" TEXT BOTH

end= /^cf=/


region= /^cf=/ /^cf=*/
```
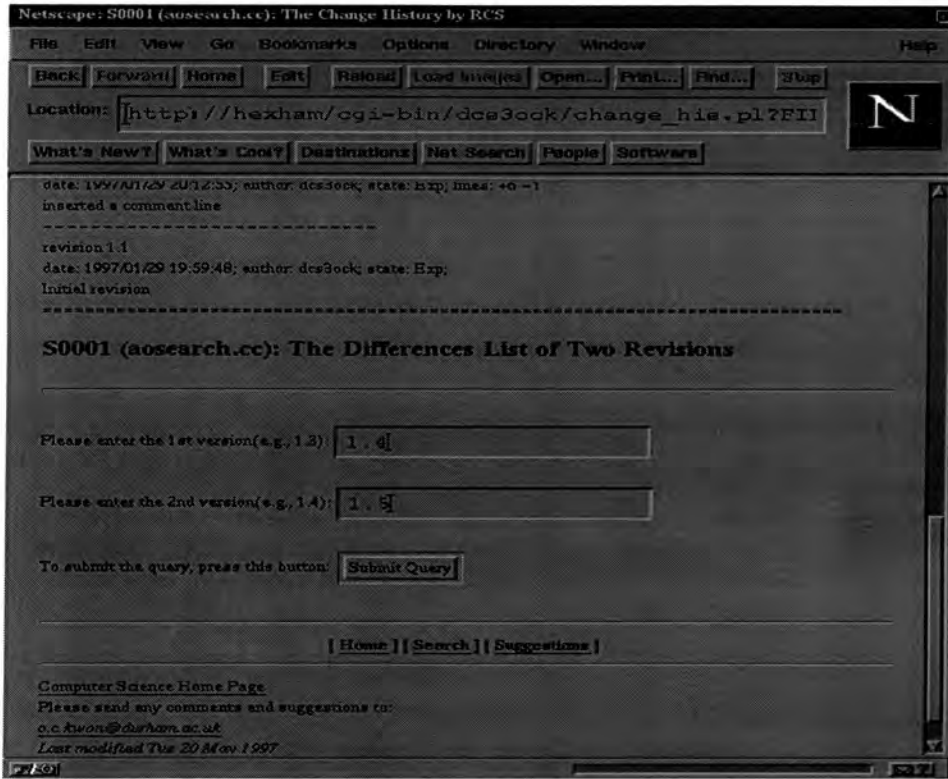
Figure A.3: A Fill-out Form for the Difference List of Two Revisions

```
cf "Component-Format" SOUNDEX LOCAL TEXT BOTH
end= /^dm=/


region= /^dm=/ /^dm=*/
dm "Domain" TEXT BOTH
end= /^mt=/


region= /^mt=/ /^mt=*/
mt "Method-Technique" TEXT BOTH
end= /^kw=/


region= /^kw=/ /^kw=*/
kw "Keyword" TEXT BOTH
end= /^rc=/
```
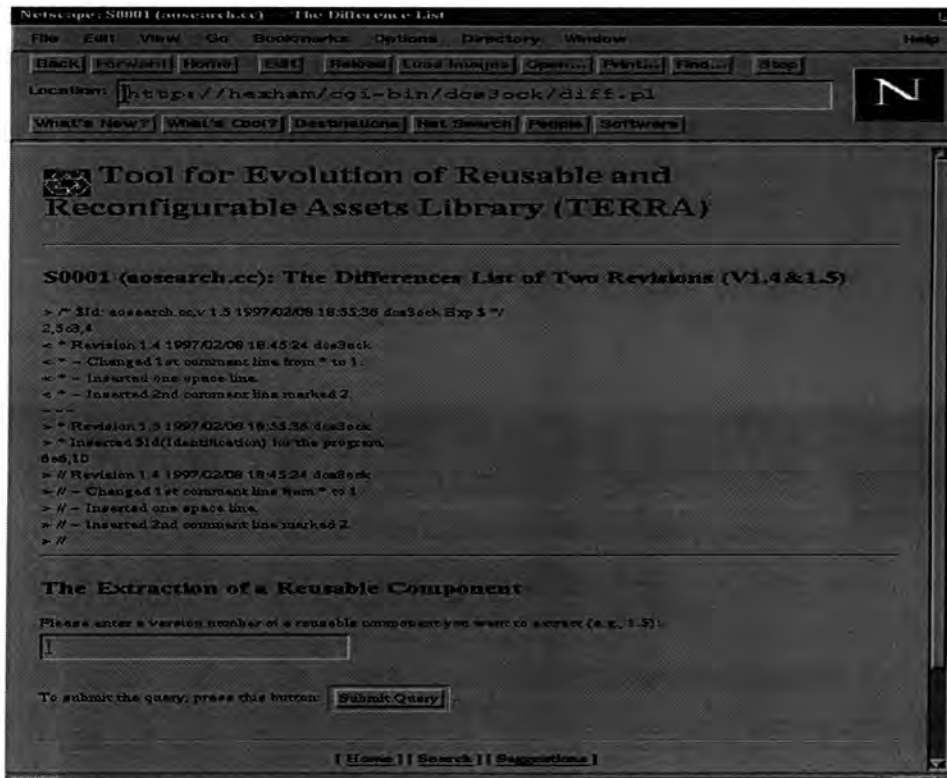
Figure A.4: The Difference List of Two Revisions

```
#region= /^abs=/ /^abs=*/
#abs "Abstract" stemming TEXT BOTH
#end= /^[A-Z][A-Z]=/
```

As shown in this file, the headline of the retrieved reusable component consists of "ci(Component ID), cn(Component Name), an(Author Name), os(Operating Systems), cl(Computer Language) and cf(Component-Format)", and provides an overview of the reusable component for potential reusers.

A.2.2 The Format Description File 'v.dfe' Used for Indexing

```
ci=Component-ID
cn=Component-Name
```

```
File  Edit  View  Go  Bookmarks  Options  Directory  Window                    Help

Back  Forward  Home    Edit    Reload  Load Images  Open...  Print...  Find...    Stop

Location: http://www.dur.ac.uk/~dcs3ock/terra/summary/reuse_S0001.

What's New?  What's Cool?  Destinations  Net Search  People  Software

Component ID=S0001
Component Version=1.2
Component name= C++ Search Class Library(AIsearch)-UNIX
Reuser Name=O C Kwon
Date of Creation=04/06/97
Reuse Type=Black Box Reuse
Summary of Modifications=None
Degree of Modification=0
Domain=Aerospace
Related Product Line=PL #2
Related Products=P #1, P #4
Other Comments=Blah Blah Blah
--------------------------------------------------------------------
Component ID=S0001
Component Version=1.3.1.1
Component name= C++ Search Class Library(AIsearch)-UNIX
Reuser Name=Hyo Seob Kim
Date of Creation=20/06/97
Reuse Type=White Box Reuse
Summary of Modifications=Modified the sorting routine to improve
performace.
Degree of Modification=10%
Domain=Business Data Processing (BDP)
Related Product Line=PL #2
Related Products=P #4, P #6
Other Comments=Blah
--------------------------------------------------------------------
Component ID=S0001
Component Version=1.4
Component name= C++ Search Class Library(AIsearch)-UNIX
Reuser Name=Rick Johnson
Date of Creation=15/07/97
Reuse Type=White Box Reuse
Summary of Modifications=Variables "X, Y", Object "ScoreResult" added
Degree of Modification=5%
```

Figure A.5: History of Reuse: Reuser's Experience Report

```
an=Author-Name
os=Operating-Systems
cl=Language
cf=Component-Format
dm=Domain
mt=Method-Technique
kw=Keyword
```

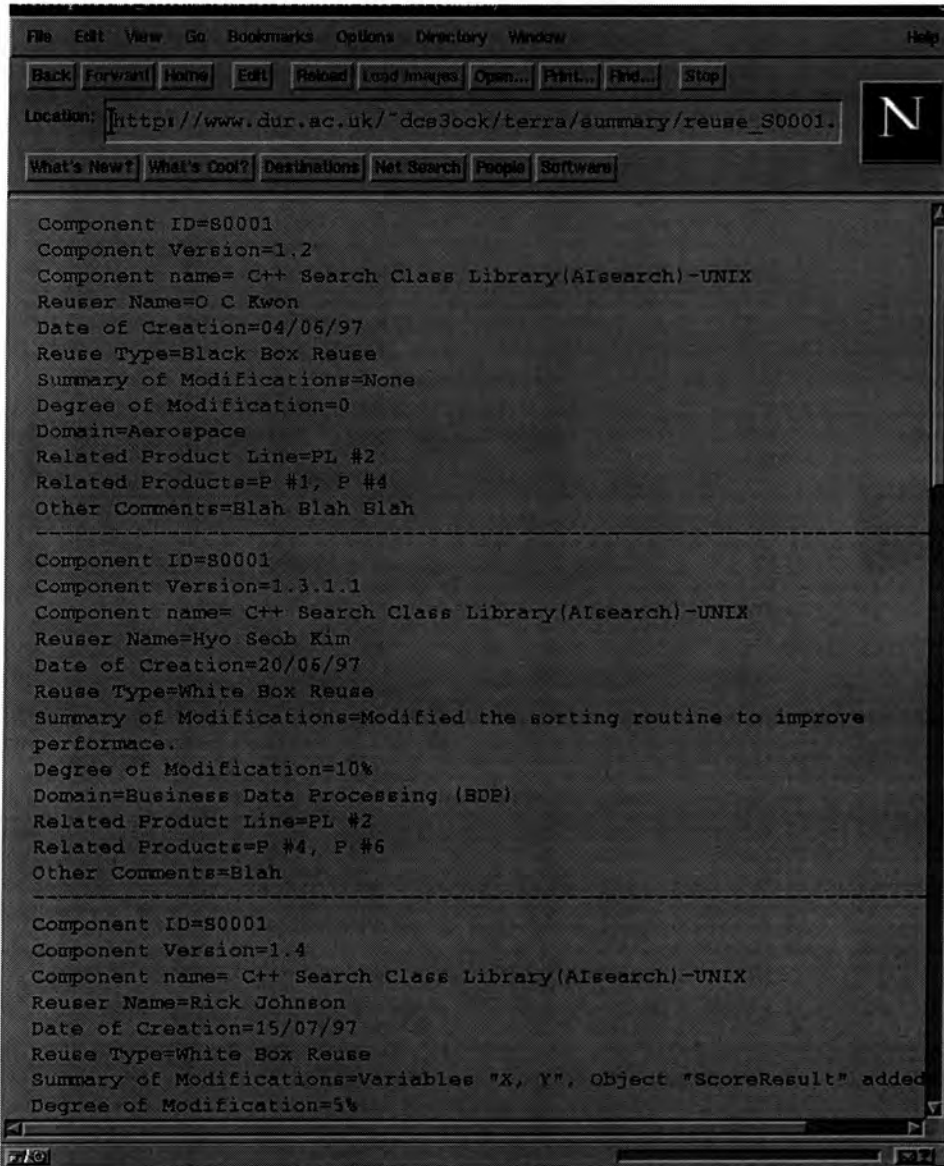As described in this file, the fields used for fast and efficient search are as follows: "ci(Component ID), cn(Component Name), an(Author Name), os(Operating Systems), cl(Computer Language), cf(Component-Format), dm(Domain), mt(Related Method and Technique) and kw(Keyword)".

A.2.3 The Index Files

The command **waisindex** generates several index files using the four files below. If **waisindex** is called by '*waisindex -d test -t fields ...*' it uses the following files:

- 'test.fmt': The format definition.

- 'test.fde': The optional format description. Plain text, which is added to the database description.

- 'test.syn': The optional synonym file contains multiple lines with synonym terms separated by spaces.

- 'test.stop': The optional stopword file contains words, one per line, which should be ignored when indexing.

The following files are generated or updated by **waisindex**:

1. 'test.src': The database description.

2. 'test.fn': The filename list. One entry for each file in the database.

3. 'test.hl': The headline list. One entry for each document in the database.

4. 'test.doc': Document table. One entry for each document in the database. It contains pointers to the filename list and the headline list.

5. 'test.cat': The catalog file. One entry for each document in the database. A human readable combination of document table with headline list and filename list. This file may be very space consuming. We can avoid generating this file if we use **waisindex** with the '-nocat' option.

6. 'test.dct': The global dictionary. One entry for each term in the default field.

7. 'test.inv': The inverted file for the default field. For each term in the database, there is a list of postings giving the documents and positions in the documents where the term occurs.

8. 'test.stop': **waisindex** might add words to the stopword file, if they occur often and would break the index.

For each field in the format description, a dictionary and an inverted file such as 'test_field_name.dct' and 'test_field_name.inv' are generated. This applies only to **waisindex** running with the '-t fields' option.

A.2.4   Some Examples for Search

- PLAIN : waissearch -d test Probabilistic Indexing
  A single or double quotation mark can be used to surround the words.

- WEIGHT : waissearch -d test 'Probabilistic < *3 Indexing'

- BOOLEAN : waissearch -d test "au=(pennekamp or fuhr) and processing"
  A single quotation mark can be used instead of a double quotation mark.

- FIELD : waissearch -d test au=Pfeifer

  If au==pfeifer is used it does not work.

- NUMERIC : waissearch -d test py==1995

  If py=1995 is used it also works.

- COMPLEX : waissearch -d test "py==1995 and (ti=(Retrieval freeWAIS) or au=pfeifer)"

  e.g. waissearch -d pro1 "ti=(software maintenance) and au=kwon"

- PARTIAL : waissearch -d test 'Pfeif*'(or "Pfeif*")

- DATE : waissearch -d test "$ed > 19930101$" (or '$ed > 19930101$')

A.3 Functions of Revision Control System (RCS)

RCS manages software libraries. It greatly increases programmer productivity by providing the following functions.

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original – previous revisions remain accessible.

   - Maintains each module as a tree of revisions.

   - Project libraries can be organised centrally, decentralised, or any way you like.

   - RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.

2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.

- RCS performs automatic record keeping.

- RCS logs all changes automatically.

- RCS guarantees project continuity.

3. RCS manages multiple lines of development.

4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.

5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.

6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one modification will not wipe out the other one.

7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.

8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.

9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a given configuration.

10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.

   - RCS provides a complete change history.

   - RCS records who did what, when and why they did it, and which revision of which module was affected.

11. RCS is fully compatible with existing software development tools. RCS is unobtrusive – its interface to the file system is such that all your existing software tools can be used as before.

12. RCS' basic user interface is extremely simple. The novice needs to learn only two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.

13. RCS simplifies software distribution if customers maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer modifications. Customer modifications can be merged into distributed versions locally or by the development group.

14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

15. RCS is implemented with reverse deltas. This means that the latest revision, which is the one that is accessed most often, is stored intact. All others are regenerated from the latest one by applying reverse deltas (backward differences). This results in fast access time for the revision needed most often.

Bibliography

[1] Aide-De-Camp Software Management System, Concord, MA, USA. *Product Overview*, 1989.

[2] W. A. Babich. *Software Configuration Management : Coordination for Team Productivity*. Addison-Wesley Publishing Company, 1986.

[3] V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, pages 19–25, January 1990.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[5] F. L. Bauer. *Software Engineering*. Information Processing 71. North Holland Publishing Co., Amsterdam, 1972.

[6] M. Ben-Menachem. *Software Configuration Management Guidebook*. McGraw-Hill Book Company, 1994.

[7] K. Bennett, B. Cornelius, M. Munro, and D. Robson. Software maintenance. In J. A. McDermid, editor, *Software Engineer's Reference Book*, pages 20/1–20/18. Butterworth-Heinemann Ltd., 1991.

[8] E. H. Bersoff, V. D. Henderson, and S. G. Siegel. *Software Configuration Management: An Investment in Product Integrity*. Prentice-Hall Inc., 1980.

[9] T. J. Biggerstaff and A. J. Perlis. *Software Reusability, Concepts and Models*, volume I, chapter 7. ACM Press, 1989.

[10] J. Blyskal and B. Hofkin. Usage scenario for the reuse library toolset. Technical report, Software Productivity Consortium, 1990.

[11] C. Boldyreff, R. Adams, et al. Development with reuse. In Patrick Hall and Liesbeth Dusink, editors, *Software Reuse*, pages 17–19. Springer-Verlag, 1989.

[12] C. Boldyreff, E. L. Burd, and R. M. Hather. An evaluation of the state of the art for application management. In *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'1994)*, pages 161–169. IEEE CS Press, September 1994.

[13] C. Boldyreff, P. Elzer, P. A. V. Hall, et al. Practitioner: Pragmatic support for the reuse of concepts in existing software. In *Proceeding of Software Engineering 1990*. Cambridge University Press, 1990.

[14] M Bott and P. Wallis. Ada and software reuse. *Software Engineering Journal*, 3(1), 1988.

[15] P. Breuer, C. Bron, et al. Design for reuse. In Patrick Hall and Liesbeth Dusink, editors, *Software Reuse*, pages 10–14. Springer-Verlag, 1989.

[16] The Technology Broker. Reaping the benefits of software evolution and reuse technologies, November 1995. A Two Day Course on Software Reuse, presented by Ericsson Software Technology, FRAMEWORKS and Q-Labs.

[17] A. W. Brown, editor. *Component-Based Software Engineering*, chapter Preface. IEEE Computer Society Press, 1996. Selected Papers from the Carnegie Mellon University (CMU)/Software Engineering Institute (SEI).

[18] J. K. Buckle. *Software Configuration Management*. Macmillan Education Ltd, 1982.

[19] F. J. Buckley. *Implementing Configuration Management : Hardware, Software, and Firmware*. IEEE Computer Society Press, 1993.

[20] T. Bull and K. Bennett. The work of the durham centre for software maintenance. Technical report, Centre for Software Maintenance (CSM), Department of Computer Science, University of Durham, October 1994. The First Research Centre on Software Maintenance established in April 1987, http://www.dur.ac.uk/CSM/.

[21] Cap Gemini Innovation. *Process Weaver User's Manual, Module 1–Initiation, and Module 3–Modelling with Process Weaver*, 1994. PW2.0.

[22] M. A. M. Capretz. *A Software Maintenance Method Based on the Software Configuration Management Discipline*. PhD thesis, University of Durham, October 1992.

[23] A. M. Christie. *Software Process Automation, The Technology and Its Adoption*. Springer–Verlag, 1995.

[24] G. M. Clemn. Replacing version-control with job-control. In *Proceedings of the 2nd International Workshop on Software Configuration Management*. ACM Press, October 1989.

[25] S. Cohen, S. Friedman, N. Solderitsch, et al. Product line identification for esc-hanscom. Special Report CMU/SEI-95-SR-024, Software Engineering Institute, Carnegie Mellon University, 1995.

[26] EUROWARE Consortium. Enabling users to reuse over wide areas (euroware) newsletters issue 1–3, 1994–1995. URL: http://www-cs.open.ac.uk/euroware/euroware.html.

[27] L. Cooprider. *The Representation of Families of Software Systems*. PhD thesis, Carnegie-Mellon University, April 1979.

[28] M. Cusumano. *Japan's Software Factories*. Oxford University Press, Oxford, UK, February 1991.

[29] C. Dabrowski and T. Kirkendall. Preliminary report on domain analysis methods. Project report, Computer Systems Laboratory, NIST, December 1992.

[30] S. A. Dart. Concepts in Configuration Management Systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 1–18, June 1991.

[31] S. A. Dart. The past, present, and future of configuration management. Technical Report CMU/SEI-92-TR-8, Software Engineering Institute, Carnegie Mellon University, July 1992.

[32] T. Davis. The reuse capability model: A basis for improving an organisation's reuse capability. In *Proceedings of the 2nd International Workshop on Software Reusability: Advances in Software Reuse*, pages 126–133. IEEE CS Press, March 1993.

[33] F. DeRemer and H. Kron. Programming-in-the-large vs programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2:321–327, June 1976.

[34] C. Desclaux and M. Ribault. Macs: Maintenance assistance capability for software a k.a.d.m.e. In *Proceedings of International Conference on Software Maintenance*, pages 2–12. IEEE CS Press, 1991.

[35] M. Dowson and J. C. Wileden. A Brief Report on the International Workshop on the Software Process and Software Environments. *ACM Software Engineering Notes (SEN)*, 10:19–23, 1985.

[36] S. H. Edwards and B. W. Weide. WISR8: 8th Annual Workshop on Software Reuse—Summary and Working Group Reports. *Software Engineering Notes (SEN)*, 22(5), September 1997.

[37] ESF/EPSOM. *Deliverable D2.1: Identification of Maintenance Activities*, March 1992. V2.0.

[38] P. H. Feiler. Configuration management models in commercial environment. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991.

[39] P. H. Feiler and G. F. Downey. Transaction-oriented configuration management: A case study. Technical Report CMU/SEI-90-TR-23, Software Engineering Institute, Carnegie Mellon University, November 1990.

[40] S. I. Feldman. Make — a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.

[41] W. B. Frakes and T. P. Pole. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 20(8):617–630, August 1994.

[42] P. Freeman. Reusable software engineering: Concepts and research directions. In P. Freeman, editor, *Proceedings of the ITT Workshop on Reusability in Programming*, pages 129–137, Stratford, Connecticut, ITT, Newport, RI, September 1983. IEEE.

[43] B. Gautier, M. Ratcliffe, and B. Whittle. Cdl: a component description language for reuse. Technical report, University of Wales, Aberystwyth, 1992.

[44] J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[45] J. Goguen. Reusing and interconnecting software components. *IEEE Computer*, pages 16–28, February 1986.

[46] J. Goguen, K. Futatsugi, and K. Okada. Parameterized programming in obj2. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 51–60, April 1987.

[47] J. Goguen and T. Winkler. Introducing obj3. Technical report, SRI International, 333 Ravenswood Ave. Menlo Park, CA 94025, 1988.

[48] P. Hall and C. Boldyreff. Software reuse. In J. A. McDermid, editor, *Software Engineer's Reference Book*, pages 41/3–41/12. Butterworth-Heinemann Ltd., 1991.

[49] P. A. V. Hall. Software components reuse - getting more out of your code. *Information and Software Technology*, January/February 1987.

[50] P.A.V. Hall. Software reuse, reverse engineering, and re-engineering. In P.A.V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*, pages 3–31. Chapman and Hall, 1992.

[51] D-R. Harjani, J-P. Queille, et al. Maintenance in a software factory—towards an integrated maintenance support environment. In *Proceedings of the ESF Seminar*, Berlin, Germany, 1992.

[52] D. Harms. *The Influence of Software Reuse on Programming Language Design*. PhD thesis, Ohio State University, 1990.

[53] W. Hegazy. *The Requirements of Testing a Class of Reusable Software Modules*. PhD thesis, Ohio State University, 1989.

[54] D. Hinley. A process modelling approach to managing software process improvement. In *Proceedings of Software Quality Management*. Computation Mechanics, 1993.

[55] D. Hinley and K. Bennett. Developing a model to manage the software maintenance process. In *Proceedings of International Conference on Software Maintenance*, pages 174–182. IEEE CS Press, November 1992.

[56] E. T. Hobbs. A uniform data model for reuse library interoperability. In *Proceedings of the 6th Annual Workshop on Software Reuse (WISR6)*, 1993.

[57] James W. Hooper and Rowena O. Chester. *Software Reuse Guidelines and Methods*. Plenum Press, 1991.

[58] W. S. Humphrey. Characterizing the software process. *IEEE Software*, 5(2):73–79, March 1988.

[59] IEEE Computer Society. *IEEE Standard for Software Maintenance*, June 1993. IEEE Std 1219-1993.

[60] IEEE Press. *IEEE Standard Glossary of Software Engineering Terminology*, 1983. ANSI/IEEE Standard 729-1983.

[61] IEEE Press. *IEEE Guide to Software Configuration Management*, 1987. ANSI/IEEE Standard 1042-1987.

[62] C. Jones. Economics of software reuse. *IEEE Computer*, pages 106–107, July 1994.

[63] E. Karlsson, editor. *Software Reuse: A Holistic Approach*. John Wiley & Sons Ltd., 1995.

[64] E. Karlsson and E. Tryggeseth. Classification of object-oriented components for reuse. In *Proceedings of TOOLS'7*. Prentice-Hall, 1992.

[65] A. Kirby, I. Sommerville, P. Rayson, et al. Versioning the web. In Reidar Conradi, editor, *Supplementary Proceedings of the 7th International Workshop on Software Configuration Management (SCM7)*, pages 163–173, Boston, USA, May 1997.

[66] P. Koltun and A. Hudson. A reuse maturity model. In *Proceedings of the 4th Annual Workshop on Software Reuse*. Centre for Innovative Technology, 1991.

[67] O. C. Kwon and C. Boldyreff. *Software Technology Analysis Task for a Software Maintenance Support Environment*. Centre for Software Maintenance (CSM), Department of Computer Science, University of Durham, April 1997. DiCE Project Report Sponsored by BT, in Collaboration with UMIST and Keele University.

[68] O. C. Kwon, C. Boldyreff, and M. Munro. An integrated process model of software configuration management for reusable components. In *Proceedings of the Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE'97)*. Knowledge Systems Institute (KSI), USA, June 1997.

[69] O. C. Kwon, C. Boldyreff, and M. Munro. Integration of a reuse process and a maintenance process within a software configuration management (scm) environment. In *Proceedings of the 8th Annual Workshop on Software Reuse (WISR8)*. Ohio State University, USA, March 1997.

[70] O.C. Kwon, C. Boldyreff, and M. Munro. Software configuration management for a reusable software library within a software maintenance environment. *The International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, September 1998. To be Published in the Special Issue on SEKE97, Knowledge Systems Institute (KSI).

[71] R. G. Lanergan and C. A. Grasso. Software engineering with reusable designs and code. *IEEE Transactions on Software Engineering*, 10:498–501, September 1984.

[72] R. G. Lanergan and B. A. Poynton. Reusable code: The application development technique of the future. In *Proceedings of the IBM SHARE/GUIDE Software Symposium*. IBM, 1979.

[73] D. B. Leblang, Chase Jr., and G. D. McLean. The domain software engineering environment for large-scale software development efforts. In *Proceedings of the IEEE Conference on Workstations*, pages 266–280, San Jose, California, November 1985. IEEE.

[74] Lie, Anund, Conradi, Reidar, et al. Change oriented versioning in a software engineering database. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 56–65. ACM Press, 1989.

[75] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 10:23–30, September 1994.

[76] D. Luckham and F. Henke. An overview of anna: A specification language for ada. In *Proceedings of IEEE CS 1984 Conference on Ada Applications and Environments*, pages 116–127. IEEE CS Press, 1984.

[77] D. Luckham, F. Henke, O. Owe, et al. Anna: A language for annotating ada programs. Technical report, Stanford University, July 1984.

[78] K. Marzullo and D. Wiebe. Jasmine: A software system modelling facility. In P. Henderson, editor, *Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments*, pages 121–130. ACM Press, 1986.

[79] M. Matsumoto. Automatic software reuse process in integrated case environment. *IEICE Transactions on Information Systems*, E75-D(5):657–673, September 1992.

[80] Y. Matsumoto. A Software Factory: An Overall Approach to Software Production. In Peter Freeman, editor, *Tutorial: Software Reusability*, pages 155–178. IEEE Computer Society Press, 1987.

[81] Y. Matsumoto, O. Sasaki, S. Nakajima, et al. SWB System : A Software Factory. In Huenke, editor, *Software Engineering Environments*, pages 305–317. North-Holland, 1981.

[82] M. D. McIlroy. Mass produced software components in : Software engineering concepts and techniques. In P. Naur, B. Randell, and J. N. Buxton, editors, *Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969. Petrocelli/Charter.

[83] M. Munro. Software maintenance, reuse and reverse engineering. In P.A.V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*, pages 573–584. Chapman and Hall, 1992.

[84] M. Paulk, B. Curtis, and M. B. Chrissis. Capability maturity model for software. Technical Report CMU/SEI-91-TR-24, Carnegie Mellon University/Software Engineering Institute (SEI), August 1991.

[85] M. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis, editors. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley Computer and Engineering Publishing Group, 1995.

[86] D. E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–71. IEEE CS Press, 1987.

[87] D. E. Perry. Version control in inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149. IEEE CS Press, 1987.

[88] U. Pfeifer. The enhanced freewais distribution. Edition 0.5 for freeWAIS-sf 2.0, October 1995.

[89] R. Prieto-Diaz. Module interconnection languages. In Peter Freeman, editor, *Tutorial:Software Reusability*, pages 117–144. IEEE Computer Society Press, 1987.

[90] R. Prieto-Diaz. Implementing faceted classification for software reuse. In *Proceedings of 12th International Conference on Software Engineering*, pages 300–304. IEEE, March 1990.

[91] R. Prieto-Diaz. Making software reuse work: An implementation model. *Software Engineering Notes*, July 1991.

[92] S. M. Przybylinski, P. J. Fowler, and J. H. Maher. Software technology transition. In *Proceedings of the 13th International Conference on Software Engineering*, page 105, 1991.

[93] M. J. Rochkind. The Source Code Control System (SCCS). *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[94] J. Rowe. *Building Internet Database Servers with CGI*. New Riders Publishing, 1996.

[95] SAIC/ASSET. *Asset Source for Software Engineering Technology (ASSET)*, 1995. URL: http://source.asset.com/.

[96] W. Scacchi and K. Naryanaswamy. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, SE-13(3):324–334, 1987.

[97] M. Simos. Organization domain modelling (odm): Formalising the core domain modelling life cycle. In *Special Issue on the 1995 Symposium on Software Reusability*. ACM Press, August 1995.

[98] M. Sitarman, L. Welch, and D. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledge Engineering*, 3(2):207–229, 1993.

[99] R. A. Snowdon and B. C. Warboys. *An Introduction to Process-Centred Environments*, chapter 1. John Wiley and Sons Inc., 1994.

[100] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.

[101] I. Sommerville and G. M. Dean. A configuration language for modelling evolving system architectures. *Software Engineering Journal (SEJ)*, 11(2), March 1996.

[102] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984.

[103] Stuttgart University, Germany. *AdaBasis*, 1995. URL: http://www.informatik.uni-stuttgart.de/ifi/ps/ada-software/ada-software.html.

[104] J. Thomas. *Module Interconnection in Programming Systems Support Abstraction*. PhD thesis, Brown University, June 1976.

[105] R. Thomson. *Automatic System Building Using a System Structure Language.* PhD thesis, University of Strathclyde, 1988.

[106] W. Tichy. *Software Development Control Based on System State Descriptions.* PhD thesis, Carnegie-Mellon University, January 1980.

[107] W. F. Tichy. A data model for programming support environments and its application. *Automated Tools for Information Systems Design,* pages 31–48, 1982.

[108] W. F. Tichy. Rcs—a system for version control. *Software Practice and Experience,* 15(7):637–654, 1985.

[109] W. Tracz. The three cons of software reuse. In *Proceedings of 3rd International Workshop: Methods and Tools for Reuse.* Syracuse University, June 1990.

[110] W. Tracz. The impact of domain analysis on software reuse. In R. Prieto-Diaz, S. Wolf, J. Cramer, et al., editors, *Proceedings of 1st International Workshop on Software Reusability,* pages 180–186. University Dortmund, June 1991.

[111] W. Tracz. *Formal Specification of Parameterised Programs in LILEANNA.* PhD thesis, Stanford University, 1992.

[112] W. Tracz. Lileanna: A parameterised programming language. In R. Prieto-Diaz and W. Frakes, editors, *Proceedings of 2nd International Workshop on Software Reuse (REUSE'93),* pages 66–79. IEEE Computer Society Press, March 1993.

[113] W Tracz. Third international conference on software reuse summary. *Software Engineering Notes,* 20(2):21–25, April 1995.

[114] University of Houston- Clear Lake, and MountainNet, Inc. *Software Market,* 1995. URL: http://rbse.mountain.net/cs/.

[115] H. J. Van-Zuylen, editor. *The REDO Compendium: Reverse Engineering for Software Maintenance.* Wiley, 1993.

[116] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Newton, MA 02164, USA, 1996.

[117] P. Walton. The management of reuse. In Patrick Hall, editor, *Software Reuse and Reverse Engineering in Practice*, pages 505–520. Chapman and Hall, 1992.

[118] S. Wartik and R. Prieto-Diaz. Criteria for comparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, September 1992.

[119] B. W. Weide, W. F. Ogden, S. H. Zweben, et al. The resolve framework and discipline- a research synopsis. *Software Engineering Notes*, 19(4):23–28, October 1994.

[120] D. Whitgift. *Methods and Tools for Software Configuration Management.* John Wiley & Sons Ltd., 1991.

[121] B. Whittle. Models and languages for component description and reuse. *Software Engineering Notes*, 20(2):76–89, April 1995.

[122] B. Whittle and M. Ratcliffe. Software component interface description for reuse. *Software Engineering Journal*, 8(6), November 1993.

[123] N. Zvegintsov. Software configuration management: Control for the software team. *Software Management News*, 11(3):13–24, May-June 1993.