Durham University

Durham E-Theses

A software maintenance method based on the software configuration management discipline

Miriam Akemi Manabe Capretz

How to cite:

Capretz, Miriam Akemi Manabe (1992) A software maintenance method based on the software configuration management discipline. Doctoral thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/6017/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

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.

A Software Maintenance Method Based on the Software Configuration Management Discipline

Miriam Akemi Manabe Capretz

Ph.D. Thesis

University of Durham

School of Engineering and Computer Science

Computer Science

October 13, 1992

2 DEC 1992

Miriam Akemi Manabe Capretz

A Software Maintenance Method Based on the Software Configuration Management Discipline

Abstract

Software maintenance has until recently been the neglected phase in the software engineering process, despite the fact that maintenance of existing software systems may account for over half of all efforts expended by a software organization. Research into software maintenance, compared to other phases of the software engineering process is rare. Moreover, it is widely accepted that current software maintenance methods and techniques are unable to cope with the complexity inherent in maintaining software systems.

This thesis is concerned with the development of a method, named Configuration Management Formalization for Maintenance (COMFORM), designed for the maintenance of existing software systems. COMFORM provides guidelines and procedures for carrying out a variety of activities performed during software maintenance. It accommodates a change control framework, around which the Software Configuration Management discipline is applied. Redocumentation is another problem tackled by COMFORM, which gathers together the documentation necessary to improve the maintainability and quality of existing software systems. This is achieved by the use of forms representing the output of each phase of a proposed software maintenance model. The information obtained by filling in forms is formalized according to a data model, which provides a common basis for the representation of the method's functionality. Finally, a prototype of COMFORM has been implemented, so that the procedures and guidelines set up by the method can be enforced and followed by its users.

Acknowledgements

Firstly I would like to thank my supervisor, Malcolm Munro, who has guided me to form many of the ideas reported in this thesis, and encouraged me throughout their development. I am also grateful to him for his comments upon the numerous drafts of this thesis.

I owe special gratitude to my husband Fernando for his constant support, assistance and patience throughout the course of this research.

Financial support for this research has been provided by grant from CAPES (Brazilian Federal Agency for Postgraduate Education).

Contents

1 Introduction **1**

1.1 Purpose of the Research 1

1.2 Thesis Aims 5

1.3 Thesis Outline 6

2 Software Maintenance of Existing Software Systems **10**

2.1 The Software Maintenance Process 10

2.2 Supporting Maintenance Technology 15

2.2.1 Reverse Engineering Research Environments 17

2.3 Documentation for Software Maintenance 22

2.3.1 Redocumentation 25

2.3.1.1 Incremental Documentation 26

2.3.2 Support Available for Software Documentation 27

2.3.3 Documentation Tools for Software Maintenance 29

2.4 Software Configuration Management 30

2.4.1 The Functions of SCM 32

2.4.2 Automation of SCM 34

2.4.3 Application of SCM to Software Maintenance 38

2.5 Summary 40

3 Software Maintenance Models **43**

3.1 Managing the Software Life-Cycle 44

3.2 Existing Software Maintenance Models 45

3.2.1 Early Software Maintenance Models 46

3.2.2 Middle Software Maintenance Models 50

3.2.3 Recent Software Maintenance Models 54

3.3 Software Process Modelling 59

3.4 Characteristics of Software Maintenance Modelling 63

3.5 Summary 65

4 Modelling Techniques **67**

4.1 The Relational Data Model 68

4.2 Semantic Data Models 70

4.2.1 The Entity-Relationship Model 70

4.3 Object-Oriented Models 72

4.4 The Object Representation Model 75

4.4.1 The Cluster Mechanism Concept 77

4.4.2 Graphical Representation of ORM 80

4.4.3 Advantages of ORM 81

4.5 Summary 82

5 COMFORM - A New Method for Software Maintenance **84**

5.1 The Objectives of COMFORM 85

5.2 The Software Maintenance Model 87

5.2.1 Change Request 91

5.2.2 Change Evaluation 93

5.2.3 Maintenance Design Specification 97

5.2.4 Maintenance Design Redocumentation 100

5.2.5 Maintenance Implementation 102

5.2.6 System Release 103

5.3 The SCM Discipline Applied to COMFORM 105

5.3.1 Software Configuration Identification 106

5.3.2 Software Configuration Control 106

5.3.3 Software Configuration Status Accounting 107

5.3.4 Software Configuration Auditing 109

5.3.4.1 Initial Evaluation 110

5.3.4.2 The Establishment of Baselines in COMFORM . . . 112

5.4 The Version Control in COMFORM 114

5.5 Final Remarks on COMFORM 118

6 Formalization of COMFORM **120**

6.1 The Modelling of COMFORM using ORM 121

6.1.1 The COMFORM Model 122

6.1.2 Consistency and Traceability in COMFORM 124

6.2 The Cluster Mechanism in COMFORM 127

6.2.1 The System Form 130

6.2.2 Scheduled Releases 131

6.3 Version Control 131

6.3.1 The Modelling of Version Control for COMFORM 132

6.3.2 The Modelling of Revisions and Variations 135

6.4 An Illustrative Example 136

6.5 Comments on the Formalization of COMFORM 139

7 The COMFORM Prototype **141**

7.1 Implementation Issues 141

7.2 The COMFORM Kernel 142

7.3 The COMFORM Framework 143

7.3.1 The CREATE operation 144

7.3.2 The GET operation 145

7.3.3 The MODIFY operation 146

7.3.4 The DELETE operation 146

7.3.5 The EFFECTUATE operation 146

7.3.6 The ESTABLISH BASELINE operation 147

7.3.7 The UNDO EFFECTUATE operation 148

7.4 The Report Generator Tool 149

7.5 Version Control 150

7.6 COMFORM in the Unix Environment 152

7.7 Summary 154

8 Use of the COMFORM Prototype **156**

8.1 The COMFORM Steps 156

8.2 Example of COMFORM Use 161

8.3 Comments on the COMFORM prototype 178

9 Conclusions **182**

9.1 Review of Work 182

9.2 The Assessment of COMFORM 184

9.3 Directions for Further Research 188

References

List of Figures

4.1 The Cluster Hierarchy for the Structured System Analysis 79

4.2 Example of use of ORM 81

5.1 The Software Maintenance Model 89

5.2 The Pattern of SMM Forms 90

5.3 The Change Proposal form 92

5.4 The Change Approval form 94

5.5 The Maintenance Specification form 98

5.6 The Module Design form 101

5.7 The Module Source Code form 103

5.8 The Configuration Release form 104

5.9 The Versioned SMM Form 116

6.1 The COMFORM Model 123

6.2 The Cluster Hierarchy of COMFORM 128

6.3 The System Form 130

6.4 The Modelling of Version Control for COMFORM 133

6.5 The Modelling of Revisions and Variations 135

6.6 An Illustrative Example of the Modelling 138

8.1 Initial information in the COMFORM object base 162

8.2 Software components in the COMFORM object base 164

8.3 A Change Proposal form 165

8.4 A Change Approval form 166

8.5 A Maintenance Specification form 168

8.6 A Module Design form 169

8.7 A Module Source Code form and an alternative 171

8.8 A Configuration Release form 172

8.9 A managerial report 173

8.10 The forms of a Configuration Release 175

8.11 The Module Design and Module Source Code forms of the PXR system177

8.12 The maintenance history of the PXR system 179

List of Tables

7.1 Parameters and options of the report generator 150

8.1 Parameters of a managerial report 174

8.2 Parameters to report the forms of a Configuration Release 174

Chapter 1

Introduction

1.1 Purpose of the Research

The last two decades have seen the promotion of software development as an engineering activity, encouraging the use of more rigorously defined software development methodologies based on sound engineering principles. Such methodologies have often emerged in response to new ideas about how to cope with complexity in software systems. So far, most research into software engineering has focused mainly on the development phases of the software life-cycle; that is, analysis, design and implementation. Until recently, however, software maintenance has been the neglected phase in the software engineering process. The literature on maintenance contains very few entries when compared to the analysis and design phases. Little research has been conducted into the subject, and few techniques and methods have

been proposed. Additionally, software maintenance has been commonly performed on the executable code, which means that after several modifications, the initial design is out of date. Consequently, later maintenance has required an extra amount of time from maintainers.

The maintenance of existing software systems may account for over sixty per cent of all efforts expended by a software development organization [16, 75]. The percentage continues to rise as more software systems are produced. As software systems age, more effort is likely to be expended on maintenance. Although most old software systems should be retired or periodically rewritten, this is not always feasible. Since most organizations are gradually becoming more dependent on existing software systems, software maintenance is crucial.

An inherent threat to maintaining a constant quality in existing software systems lies in the fact that they are prone to changes. Besides, every change to software carries with it the potential for introducing errors, or creating side effects which propagate errors. Since changes in software systems are inevitable, mechanisms for making, evaluating and controlling modifications must be considered. In order to be cost-effective, a better approach to software maintenance is imperative; the software maintenance process must be made simpler and more productive. To simplify the maintenance process, software change must be controllable, so that software systems avoid falling into chaos. A more productive process can undoubtedly be obtained by taking an organized and structured approach to maintenance.

Despite increasing recognition that maintenance is a major problem during the life-cycle of software systems, and the acknowledgement of the high costs of maintaining software systems, there are clearly gaps in the field of software maintenance. So far,

there is no generally accepted method which systematizes the software maintenance process; neither is there an integrated set of tools which would help to tackle the problem of controlling software changes under a systematic scheme.

This research is aimed at the creation of a method named Configuration Management Formalization for Maintenance (COMFORM). This method provides guidelines and procedures for carrying out the maintenance process, while establishing a systematic approach for the support of existing software systems. COMFORM takes the Software Configuration Management (SCM) discipline into account, since this discipline imposes a set of procedures and standards for managing evolving software systems. The application of the SCM discipline contributes directly to software quality by identifying and controlling change, assuring the change is being properly implemented, and reporting the change to others who may have an interest. A change control framework has been established in COMFORM in order to preserve software quality. The Software Maintenance Model (SMM) institutes this framework, which aims to systematize the software maintenance process by specifying the chain of events and the order of stages that a change has to go through.

Redocumentation is another problem tackled by COMFORM, which gathers together the documentation necessary to improve the maintainability of existing software systems. During software maintenance, documentation should provide an abstract representation of the operational software system, which guides the decisions that affect the system's evolution [15]. In order to obtain the documentation required for the method, the following imperatives have been stated: standardization of the information to be stored, in order to guarantee its completeness and preciseness; and the uniformity of document structures of the software systems to be maintained must be upheld. To satisfy these demands, information is obtained by filling in

forms. The intention of creating forms is to make the maintainers' task one of filling in forms for documentation, instead of defining their own document structures. The forms represent the results of SMM phases, which allow a methodical approach to be taken towards the establishment and control of traceability throughout the maintenance process.

The information obtained by filling in forms has been formalized according to a data model, which provides a common basis for the representation of the method's functionality. The conceptual model for COMFORM has been defined using the data model termed Object Representation Model (ORM) [111]. ORM has been used because of its enhanced semantic capabilities, which provide the necessary generality and standardization for software representation.

Experience with software development has shown that effective use of a particular method is achieved by computer-aided support. In the last few years, therefore, environments and tools supporting (and enhancing) software development methods and procedures have emerged. This technology, known as CASE (Computer-Aided Software Engineering), allows system analysts to document and model a software system from its initial user requirements through the design and implementation stages. Moreover, by using an integrated set of tools, CASE environments encourage both conformance to an underlying method and the application of tests for consistency and completeness. Given that a reasonable solution for the problems of software maintenance is to extend good software development practice into maintenance activities, the ultimate aim of COMFORM is to provide support within a software maintenance environment. By these means, the procedures and guidelines set up by COMFORM can be enforced and followed by maintainers, and software quality will not deteriorate.

1.2 Thesis Aims

The novel contribution to software maintenance of the approach taken by COMFORM is its systematic support of existing software systems through the use of the SCM discipline, whilst incrementally documenting them. A survey of current literature failed to reveal any other work which tackles this problem as COMFORM does. The main aims of this thesis are:

- The development of a new method - COMFORM - which provides guidelines and procedures for the maintenance process by applying the SCM discipline. This method improves the balance between code and higher level abstractions after any change in the software system has been made. As a result, a reliable and easily understandable documentation of an existing software system can be incrementally obtained whilst it is being maintained.
- The definition of a Software Maintenance Model (SMM), the phases of which institute a change control framework to monitor changes. The SMM aims at systematizing the software maintenance process by specifying the chain of events and the order of stages that a change has to go through.
- The creation of forms to reflect each of the SMM phases. These forms constitute a methodical approach to software maintenance driving towards the establishment and control of traceability throughout the SMM phases.
- The production of a number of reports to aid project managers and maintainers, by providing updated information about the software system's current status, as well as about maintenance history.

- The formalization of COMFORM by the use of ORM. This formalization provides the representation of information and semantic characteristics, facilitating the method's realization.
- The implementation of a prototype which forms the foundation of a software maintenance environment, consisting of a framework which provides automated support for the concepts and capabilities of COMFORM.

1.3 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 expands on the general background of software maintenance and the necessary documentation, as well as introduces the SCM discipline to maintain existing software systems. The first section introduces the concepts, problems and categories of software maintenance. Section 2 concentrates on the supporting maintenance technology. In Section 3, the documentation necessary to maintain existing software systems and the approach of incremental documentation are presented. Section 4 looks at the SCM discipline, which has been applied mainly to the software development process, as another important factor in helping the maintenance of existing software systems. A summary of this chapter can be found in Section 5. The conclusion which can be drawn from this chapter is that up to the present time, there has been no method for maintaining existing software systems similar to the one proposed by this thesis, i.e., incrementally documenting existing software systems while maintaining them under SCM control.

Chapter 3 describes existing software development models. Section 1 examines the use of models to manage the software life-cycle. In the second section, a series of software maintenance models, aimed solely at maintaining existing software systems, is presented and compared. The third section introduces the notions and desirable characteristics of software process modelling for software development and maintenance, so as to provide a basis for the comparison of the research in this thesis with previous investigations. Section 4 identifies specific points of the software maintenance modelling concept considered in this thesis. A summary of this chapter is presented in the fifth section.

Chapter 4 provides a description of various modelling techniques. The traditional modelling techniques are reviewed, along with their characteristics and applications to model software systems. These data models are presented so as to introduce the Object Representation Model (ORM), putting it in the context of data models. Section 1 describes the relational data model. Semantic data models are reviewed in Section 2, with emphasis given to the well-known Entity-Relationship model. Section 3 looks at the main characteristics of object-oriented models. ORM is then described in Section 4, along with its main characteristics, which contributed to the modelling of the method proposed in this thesis. The last section concludes with a summary of the chapter.

Chapter 5 contains a detailed description of the proposed method (COMFORM) for the maintenance of existing software systems. Section 1 examines the objectives of COMFORM. In Section 2, the Software Maintenance Model (SMM), together with its phases and corresponding forms, is described. The role of the SCM discipline, and the application of each SCM function to COMFORM are discussed in Section 3. Section 4 details the version control applied to the forms. Lastly, the

chapter concludes with observations on the proposed method.

Chapter 6 examines the process of formalizing the method (presented in Chapter 5) using ORM. Section 1 details the formalization of the method. In Section 2, the application of the cluster mechanism of ORM is discussed in order to clarify its use and to help in the implementation of some basic concepts of COMFORM. Section 3 describes the modelling of version control for COMFORM. An illustrative example is given in Section 4, which helps to visualize the formalization of COMFORM. The chapter ends with a summary of the formalization of the method.

Chapter 7 describes the implementation of the COMFORM prototype. The main purpose of the prototype is to provide an automated framework into which other tools can be incorporated, so that this framework represents the basis of a software maintenance environment, supporting the method described in Chapter 5. The first section discusses the overall issues related to the implementation of the prototype. Section 2 outlines the COMFORM kernel, which is central to integrating tools into the framework. In the third section, the implementation of the COMFORM framework, along with its operations are described. Section 4 presents the report generator tool, which provides a number of reports for project managers and maintainers. Section 5 outlines the basic features of version control implemented within the prototype. Section 6 then goes over the issues of linking the COMFORM prototype with other tools in the UNIX environment. The chapter concludes by pointing out the performance of the quality assurance checks in the COMFORM prototype.

Chapter 8 discusses the results of applying COMFORM with the purpose of maintaining an existing software system. In the first section, the steps to be followed

when applying COMFORM for the purpose of maintaining existing software systems are outlined. Section 2 gives an example of COMFORM use by maintaining an existing software system named PXR (a context-sensitive cross-reference tool). The chapter ends with observations about the COMFORM prototype.

Chapter 9 presents the conclusions of this thesis. In Section 1, a review of the work is described, reiterating the aims of this research. Section 2 provides an assessment of COMFORM, analysing the work which has been carried out in this thesis. The last section of the chapter discusses possible directions and ideas for future research, and concludes with some final observations.

Chapter 2

Software Maintenance of Existing Software Systems

This chapter contextualizes the research of this thesis by describing the software maintenance process, the supporting maintenance technology, documentation for software maintenance, and the software configuration management discipline.

2.1 The Software Maintenance Process

Software maintenance has been defined by the IEEE [57] 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.

Software maintenance consists of a series of activities required to keep a software system operational and responsive after it is accepted and placed into production. In between, a variety of activities involving maintainers, quality assurance and configuration management personnel must be planned for, coordinated and implemented [4].

The current concern about software maintenance is that it is recognized as the most expensive phase of the software life-cycle [75]. In addition, the quality of code repairs and updates is often poor and this compromises the software system reliability and performance. Although many activities related to the maintenance and development of software systems are similar, software maintenance has unique characteristics of its own [4, 8, 106], as detailed below:

- Software maintenance is performed on an existing software system. Any changes introduced to a software system must conform to or be compatible with its architecture, design and code constraints.
- Software maintenance typically requires that a programmer spend a significant proportion of his or her time attempting to understand how the program is constructed and how it functions.
- Software maintenance is usually open-ended, continuing for many years (as long as it is economically possible), in contrast to software development, which is undertaken within a timescale and to a budget.

- During software development, test data is created from scratch. Software maintenance can use this existing test data and perform regression tests, or alternatively, create new data to test only the changes and their impact on the rest of the software system adequately.

Software maintenance activities are commonly classified into four categories [10, 81, 88], based on the areas first defined by Swanson [107]. These categories are:

- **Perfective Maintenance:** enhancing the software system by altering its functional behaviour, which resulted from a change in the original intent or requirements.
- **Adaptive Maintenance:** changing the software system in response to a change in the data environment (system input and output formats), or in the processing environment (either hardware or support software).
- **Corrective Maintenance:** diagnosing and correcting errors which cause incorrect output or abnormal termination of the software system.
- **Preventive Maintenance:** updating the software system to anticipate future problems; this entails improving the quality of the software and documentation, or other software quality factors. Modifications in this activity do not affect the functional behaviour of the software system. Aspects of reverse engineering can be considered as part of this category.

Not all modifications strictly belong to one category or another. For instance, corrective maintenance may also require enhancement (perfective maintenance) of a sub-system. Similarly, a sub-system, because of an inability to correct a persistent fault in it, may be re-designed to improve maintainability (preventive maintenance).

Although this research seems to be directed towards preventive maintenance, (i.e., it aims to improve the documentation and maintainability of existing software systems), it is also very much related to the other maintenance categories (adaptive, corrective and perfective maintenance).

Many problems associated with software maintenance can be traced to deficiencies in the way software systems have been defined and developed. Lack of control and discipline in the initial phases of software development nearly always translates into problems during the maintenance phase. As this lack of control and discipline persists in the course of maintaining software systems, the structure and maintainability of such systems tend to deteriorate, often making them more complex and difficult to maintain next time. The cost of failing to design software systems for maintenance is very high. Such systems become so fragile that maintainers and their managers are reluctant to change them. Any change may have unforeseen consequences, often causing problems in other parts of the system, annoying users and consuming precious software personnel time [80].

The necessity of performing software maintenance activities on a software system that has not been designed with maintenance in mind is arduous. These software systems are usually, for example, poorly documented, with an uncomprehensible structure, and with the data representation embedded in the program code. One consequence of this lack of maintainability during software development is loss of traceability, i.e., the ability to identify technical information which relates to a software error detected [98]. Moreover, to make changes to a software system, maintainers normally need to understand the components of such a system, e.g., the contents and purpose of modules in the software system, and the historical context in which the modules were developed. In old existing software systems, such information is

usually unavailable or out of date, and this situation deteriorates even more after maintenance is performed on those systems.

The net result of continual changes is that software tends to increase in size, and its structure tends to degrade with time. For example, Harrison and Magel [53] show that the average program grows by 10% every year, doubling in size every seven years. Therefore, it is important to increase the maintainability of software in order to cope with its increasing lifespan.

It has been argued that new development techniques will reduce maintenance costs in the future. However, it has also been shown that software systems must evolve if they are to continue to be useful, and thus, perfective maintenance work will always be a vital activity. Lientz and Swanson claim that user enhancements account for 42% of all maintenance work, and that only 20% is corrective maintenance. Furthermore, user demands for enhancements and extensions were identified as the most important management problem area. Therefore, whilst new development techniques might reduce corrective maintenance, they will have little effect on the most expensive sub-task, perfective maintenance. Another interesting observation shown by this survey is that most corrective and adaptive maintenance work is considered obligatory, while the other two types of maintenance are mostly optional. Therefore, nearly half of all maintenance work is optional. From this, it follows that one way of minimizing the cost of software maintenance is to investigate perfective and preventive maintenance activities more carefully. Martin and McClure [80] suggest schemes such as change request logs and formal change procedures, so that fewer and more thoughtful change requests can be generated.

Currently, research into software maintenance tools and environments is being undertaken, but it is far less common than research into software development. Hitherto, no environment or tools have succeeded in integrating the diversity of maintenance tasks in a coherent way. Automated support is usually restricted to a single task of the maintenance process. There is a lack of automated maintenance support that could help maintainers during the whole software maintenance process, from change request to system release.

It is interesting to note that many projects concerned with software development claim that they will also assist maintenance. This is debatable as environments devoted to software maintenance have particular characteristics of their own. One such characteristic, for instance, is the capacity to capture the information of existing software systems into the environment. The various approaches of tackling such a problem are discussed in the next section.

2.2 Supporting Maintenance Technology

Old software systems have not usually benefited from modern software technology such as CASE (Computer Aided Software Engineering). Therefore, the use of software maintenance techniques, such as re-engineering, has been promoted as the answer to many of the problems of maintaining existing software systems. The basic idea underlying re-engineering [8] is that design and specification information is extracted from the source code, so that it may be used to gain insight into the purpose of the software system, or to replace part or all of the software system with modern software technology.

Software re-engineering may also be used for restructuring systems, extracting reusable parts and providing new views of software and its documentation. The hope is that the maintainability and adaptability of existing software systems can be improved. The basis of all approaches to re-engineering is to try to make existing software systems easier to maintain. Unfortunately, re-engineering lacks a standard terminology and as such, different people often use different terms to cover the same basic concepts (and sometimes, the same names for different concepts). Chikofsky and Cross II [29] have attempted to clarify terms such as reverse engineering, redocumentation, design recovery and restructuring, as they are described below.

Reverse Engineering

Reverse engineering is the process of analysing a software system to identify its components and their inter-relationships creating representations of the software system in another form or at a higher level of abstraction. It is the part of the maintenance process which improves understanding of software systems and its structure, by giving a sufficient design-level view to aid maintenance, strengthen enhancement, or support replacement, given at worst the source codes. Two sub-areas of reverse engineering widely referred to are redocumentation and design recovery.

Redocumentation aims to recover the documentation of a software system. It involves experienced programmers creating documentation by analysis of the source code and the recovery of useful documentation from the original documentation.

Design recovery [14] recreates design abstractions from a combination of code, existing design documentation (if available), personal experience and general knowledge about problem and application domains. It should reproduce the information required in order for a person to fully understand a program; namely, what the

program does, and how and why it does it.

Restructuring

Restructuring is the transformation of one representation form to another at the same relative abstraction level, while preserving the software system's external behaviour (functionality and semantics). It is often used as a form of preventive maintenance to improve the physical state of an existing software system, with respect to some preferred standard. It may also involve adjusting the software system to meet new environmental constraints that do not involve reassessment at higher abstraction levels.

2.2.1 Reverse Engineering Research Environments

Currently the number of available stand-alone tools, such as static analysers and control flow restructurers, are increasing in number. However, these tools do not support any method for reverse engineering and are not designed to be integrated with other tools. Some projects, however, have the objectives of supporting methods and tools for reverse engineering, rather than simple restructuring operations. Nevertheless, the approaches adopted by the projects differ significantly. The ASU software maintenance environment [33], for instance, is being developed to support both managerial and technical maintenance tasks. It operates on the Pascal code and provides some facilities to understand, document and analyse the code for ripple effects.

Lano and Haughton [72] developed a method based upon the language Z++ to

support the use of formal methods in software maintenance. The method is centered on the maintenance of the specifications and not upon the source code. Such formal specifications can be created, either from user requirements (forward engineering), or by reverse engineering an application. The abstraction transformations are recorded during the reverse engineering process, building up the documentation. Change requests are translated and implemented into change requests to the specification, so that the application implementation can be generated together with revised documentation.

SOFTMAN [30, 31] takes a different approach. Although it is designed to support forward engineering, it also provides support to existing software systems by using reverse engineering. A noteworthy characteristic of SOFTMAN is that it supports incremental verification and validation of software correctness across all life-cycle activities. In SOFTMAN, a software system is correct if all of its life-cycle descriptions are traceable, consistent and complete. Its method of maintaining existing software systems consists of importing them into the SOFTMAN environment to provide reverse engineering and subsequent forward engineering support. This approach to reverse engineering is also being taken by other large projects in the area of software maintenance. Some of these projects are further discussed below. They aim to provide method and tool support for reverse engineering, and they place considerable emphasis on a single integrated representation of the original system in a system database.

REDO

The aim of REDO [8] is to assist software engineers in the maintenance, restructuring and validation of large software systems and their transportation between

different environments. The objective is to articulate a theoretical framework for doing this and to develop the necessary methods and prototype tools. REDO is aimed, in the first instance, at the maintenance of Cobol data processing systems and scientific applications written in Fortran. The major focus and contribution of the REDO project is to provide a tool set and environment for the reconstruction of software. The purpose of reconstruction is to take the software as it stands, and then to produce a more maintainable version. The REDO tools are integrated around a central database containing the application itself and all related information, including documentation and test data. A uniform user-interface to the toolkit has been developed. Applications are translated into an internal intermediate language, upon which the tools operate. The tools and methods are thus independent of the target language, and may be applied to other languages by building suitable intermediate language translators.

MACS (Maintenance Assistance Capability for Software)

The strategy of MACS [8, 35] is to provide a maintenance assistance system, in the form of a tool set which comprises expert system tools that provide the assistance to a maintainer. MACS covers the phases of reverse engineering, impact analysis and change management. It is intended to recover design specifications, given only source codes, so that functional specifications of the software can be recreated. From these, a new implementation of the application program can be achieved using modern software engineering methods. The MACS approach to software maintenance has three main characteristics. First, the design and structure of an existing software system is extracted using reverse engineering and is represented in a language-independent formalism called dimensional design. Second, the way in which software maintenance is undertaken is addressed by integrated front-end

tools holding knowledge about the maintenance process and expert maintenance behaviour. The design of the user interface is strongly influenced by human factor analysis. Finally, the MACS system also attempts to capture design decisions and their rationale.

ReForm (Reverse Engineering using Formal Transformations)

The aim of the ReForm project [44, 113, 114] is to develop a tool called the Maintainer's Assistant. The main objective of this tool is to develop a formal specification from old code. It is concerned specifically with reverse engineering existing software systems in order to bring them to a state in which modern software engineering techniques can be applied. The objective is to enable software, written in low-level procedural languages (in particular, the code for CICS written in IBM assembler), to be expressed in terms of non-procedural abstract specifications (e.g. Z) via a process of applying formal transformations. The key feature of this is the transformational approach: using a wide-spectrum language and verified transformations, in order to simplify and abstract the low level code.

All three projects (REDO, MACS and ReForm) use a common structure of a single repository in which all software items are stored, and where they are then accessed for manipulation by tools. All three share the same presupposition, namely that a completely automatic approach to reverse engineering is impossible, though simple automatic tools can help at the tactical level.

Although the REDO project provides a contribution to the task of maintaining existing software systems, it lacks a discipline to control the reconstruction process of the software. REDO takes a top-down procedure to reverse engineering, by focusing on the process and method, and then implementing an environment and tool set to

support the method. Its key aspect is the integration of the tool set through the central repository.

The approach of the MACS project differs from REDO and ReForm, as it provides automation through an expert system style, and aims to provide an assistant, instead of a fully automatic tool set. It encapsulates knowledge about both the application domain and implementation, and the expertise of software maintainers. In this sense, similar work which can be related to MACS includes SOFTM [83, 84], which is limited to error localization and treatment, and the Desire system [14]. MACS provides some control for the reverse engineering process, since one of the layers presented in its architecture comprises a comprehensive configuration management system. Such functionality supports the changes during the maintenance process.

The approach of ReForm project is also different from the others, in that it consists of an interactive system for maintaining programs, which is based on program transformations. The transformations derive the specification of a program section, presenting the program in a different but equivalent form as an aid to program analysis and for general restructuring functions.

From the projects discussed in this section, it can be seen that the objective of the software maintenance environments is mainly reverse engineering existing software systems, to subsequently handing them over to forward engineering support. While reverse engineering may increase the maintainability of these systems, it may also incur unacceptable expenditure. When dealing with software systems having a predicted long maintenance life, this technique may be economically viable, but it seems unlikely to be appropriate for every kind of existing software systems. Therefore, alternative techniques to software maintenance need to be explored, so that less

expensive methods of maintaining existing software systems can be developed.

2.3 Documentation for Software Maintenance

Documentation is a critical and controversial issue in software maintenance. The Lientz and Swanson survey [75] identifies quality of software documentation as the most significant technical problem. A survey by Chapin [26] of personnel close to software maintenance work showed that they perceived poor documentation as the biggest problem in this field.

Maintenance documentation provided by developers is, or should be, the major source of system information for maintainers. If the documentation is inadequate, the maintainer must use less convenient sources, or sacrifice the quality of the modification. This increases the risk of introducing an error by the modification, and causes system maintainability to deteriorate. The documentation should be accurate, usable and trusted by the maintainers. Outdated documentation is not only useless, but may also complicate and confuse the already difficult maintenance task [80].

Documentation has always been given a low priority status, compared to other activities in software development. Documentation is nearly always left to the end of a project, and as projects invariably run late, the amount of documentation originally planned is reduced, or even completely cancelled. It is not unusual to find that the only documentation concerned with the design of the software system are comments within the source code itself. Moreover, software documentation is usually written

by developers who do not understand the maintenance process. Therefore, document structure does not always provide enough visibility for maintenance concerns [52].

The controversy over software documentation is really concerned with the type of documentation that is needed, rather than the question of whether or not any documentation is needed at all. Not only the source code of software systems, but also their documentation, must be maintainable. Producing voluminous amounts of detailed software documentation, requiring a major update effort each time a software component is modified, can only compound the maintenance burden.

Unlike the documentation that should be generated during software development, documentation during software maintenance does not mean the generation of a complete set of documents of a software system [15]. As far as documentation is concerned, there are several differences between the processes of initial software development and software maintenance. In the initial software development, there is more freedom in making implementation decisions. During most of this period, no source code is available, and there may be an imperfect understanding of how the software system should operate. In this case, the primary function of documentation is to make concepts clear and to communicate decisions. In the case of maintenance, there is considerably less flexibility. All design decisions must be integrated with the existing (and evolving) software system; both designers and users have a better understanding of their needs. Thus, at this stage, the documentation provides an abstract representation of the operational software system, which guides the design decisions that affect its evolution.

The most significant information required for maintenance is a description of what the software system does, and which software components cause it to do this. This

depends heavily on a high-level description of the structure of the software system which assists in locating specific information. Documentation should also explain why the users want the software system to do what it does, so maintainers can serve the real needs of users.

In addition, knowing how a software system has evolved during its development and maintenance is very useful information to the maintainer. Because the value of historic software development information has not been recognized, it is rarely kept. However, it can greatly simplify the maintenance task. As well as clarifying user needs and development principles, this practice can help indicate the portion of the software system affected by particular decisions [82]. Understanding the original design intention may guide the maintainer in the process of choosing ways to modify the code that do not jeopardize system integrity. Also, knowing the parts of the system that developers considered the most difficult may give the maintainer a first clue to where an error might lie [80].

Even within the maintenance process, documentation is not often produced, despite the maintainers' view that it is useful. Thus, the documentation, when carried out after the maintenance activity, is often inadequate in relation to the magnitude of the system change. The documentation of software evolution is an important aspect of software maintenance. Problem tracking and reporting is central to documenting software evolution [50]. They are key aspects of recording changes to large software systems. Large software systems which evolve over a long period of time magnify the need for organized change tracking, due to the number and extent of the changes made. Change histories provide important information about the reasons behind the changes to software systems. In a study conducted by Collofello and Buck [34], it was found that more than 50% of errors or faults were introduced by previous changes;

the record of past changes is clearly a major contributor to software maintenance. Using these records, the original cause of errors can be traced, allowing the re-design of the original change, or at least, a better understanding of the cause of the problem.

In a more immediate sense, collection of data about the maintenance of a software system increases the visibility of the maintenance process and provides information to the senior management. A record of changes, and the reasons for their institution, forms a permanent history of experience gained by maintainers.

2.3.1 Redocumentation

It is not always convenient to completely reproduce the design documentation from the source code, as it has not yet been established whether or not this is the most suitable documentation for software maintenance. In general, high-level documentation explaining the overall purpose of the software system and describing the relationship of its various components is the most useful [80]. Redocumentation for existing software systems can be a cheap yet effective productivity aid, especially when the process is undertaken incrementally and under a quality assurance procedure.

Many maintenance teams are forced into redocumentation because the documentation that is supplied with the software system they have to maintain is inadequate or nonexistent. One way of undertaking redocumentation is to reproduce the design documentation from the source code. Unstructured and inadequately documented software systems, which prove difficult to maintain, can be redesigned using modern programming practices. This technique is usually considered uneconomic on all but

the smallest software systems. However, if this path is taken, it would be advantageous to use a documentation-support environment that managed the life-cycle documentation set in order to have full control of the process [40]. These environments are primarily aimed at those designing new software systems, but they would also be of use in this approach to redocumentation. They enforce standards on the documentation and allow all the development documentation to be centrally located, providing easy access and updates.

2.3.1.1 Incremental Documentation

Since it may not be economically viable to reproduce the whole design documentation, it is important to try to identify techniques that allow redocumentation to take place gradually during the maintenance activity itself. A system for incrementally redocumenting old software systems in this way has already been described by Fletton and Munro [41]. Foster and Munro [43] argue that redocumentation of this type encapsulates some of the knowledge acquired by maintainers of a large software system, which could otherwise be lost when staff leave.

If a redocumentation system for capturing the knowledge gained by maintainers while analysing source code is to be set up, a number of key requirements, as described by Fletton [40], have been stipulated. One of these requirements is incremental documentation, i.e., the ability to build up documentation of a software system over a period of time in an incremental manner, without the need to document the complete system in one step. This is an important requirement of a redocumentation system, as it allows the documentation to be produced as code is examined during the day to day software maintenance process, and not as an activity in its

own right.

Another benefit is that only the code which is analysed by maintainers is documented. No time should be spent documenting code that is in a stable state and may never be examined or modified. It has often been said that the 80/20 rule applies to software maintenance; 80 per cent of the time is spent on 20 per cent of the code [43]. Therefore, it is unproductive to document a complete system during software maintenance. The maintenance personnel play a key role in this process by providing information about the existing software system. The information they provide includes the various assumptions that are made during software implementation, and other semantic information that has not been recorded and is difficult, if not impossible to obtain automatically with tools. By adopting this method, the maintenance staff should be able to enter those details they judge to be important for the maintenance of a particular software system.

2.3.2 Support Available for Software Documentation

Over the past years a number of software documentation environments which support the production, management and use of textual and graphical documentation during all phases of the software life-cycle have been discussed in the literature of this field of study. Most of these documentation environments provide facilities to support traceability, central storage of all project documentation, easy access and update, and the enforcement of project-wide standards on the structure of documentation. The SODOS [55, 56] and DIF [47, 48, 49] systems are examples of such environments; they concentrate on the production of conventional life-cycle documentation during the development of a software system. Thus, they are of little

use to maintainers faced with a completed software system which has little or no existing documentation.

SODOS (Software Documentation Support)

SODOS [55, 56] supports the definition and manipulation of documents used during software development. It enforces standards on the documentation and allows all the development documentation to be centrally located, providing easy access and updates. The SODOS system maintains a relationship between code and its description, so that it is easy to go back and forth between them. Documentation provided in this form is likely to be a major factor in reducing the cost of software maintenance. Nevertheless, SODOS is of limited use for the retrospective documentation of existing software systems during software maintenance. It would be necessary to redocument the whole system before any gains could be achieved in the maintenance phase. This is usually prohibitively expensive.

DIF (Document Integration Facility)

DIF [47, 48, 49] is a software hypertext system for integrating and managing the documents produced and used throughout a software life-cycle. DIF departs slightly from the other environments in that it has the additional aim of integrating documents within and across several projects into a single environment. Hypertext is used to provide traceability between life-cycle objects (such as requirement analysis, design code and tests). Like SODOS, DIF manages life-cycle documents in an object-oriented fashion. Its revision mechanism is limited to letting the user define the revision numbers for part of the documents stored.

In order to ensure that all the projects have a standard document structure, each

document produced by DIF is defined as a form, which has a tree-structured organization of basic templates to be instantiated with project-specific information. Such forms provide a rudimentary way of defining the software process which is to be followed by the projects. The concept of forms, used to develop the research described in this thesis, is similar to the concept of forms and basic templates in DIF. However, DIF does not provide support to the maintenance process or to redocumentation in the same way as the work proposed in this thesis does, as will be seen in Chapter 5.

2.3.3 Documentation Tools for Software Maintenance

There are a number of tools available which claim to satisfy the documentation needs of software maintenance. The advantages of these tools are that they are inexpensive to operate and the documentation produced is easily kept up-to-date. Most take the form of static analysis tools, producing a series of reports [27, 71, 99, 103]. Examples of the documents produced are: control/data flow charts, cross-reference listings, metric reports, call graphs and module hierarchy charts. All this information is of significant use to the maintainer when becoming familiar with the structure of a software system and in navigating around its components during maintenance investigations. What they fail to do is to provide any insight into why particular structures are used, or why certain design routes were taken. This knowledge can only be recovered by eliciting information from the original designers, or by detailed examination of the source code by maintainers.

At present, the number of choices available to a maintenance team faced with redocumenting large software systems is limited. A documentation system for recording

the knowledge obtained from the source code along with the cross-reference information has been described by Foster and Munro [43]. Contributions to support redocumentation during software maintenance (in the form of incremental documentation but also involving hypertext technology) can be seen in the works described in Fletton [40], Fletton and Munro [41], and Kenning and Munro [63, 64].

2.4 Software Configuration Management

Software Configuration Management (SCM) [5, 13, 22] is an important element of software quality assurance. The purpose of the SCM discipline is to manage change throughout the software development and maintenance processes. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual software configuration items and various versions of the software, the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration [88]. It is a useful discipline to the project manager, allowing this complex task to be executed in a well-organized way, and maximizing productivity by minimizing mistakes [5].

The SCM discipline has been defined by Bersoff, Henderson and Siegel [13] 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.*

In the SCM context, a component of a software system is called a Software Configuration Item (SCI). A SCI is a part of the software system which is treated as a unit for the purpose of SCM. SCIs may be decomposed into other SCIs; they may also be modified, thus creating versions of the original SCIs. The range of SCIs which SCM must manage is very wide and includes source code, executable code, user and system documentation, test data, support software, libraries, specifications and project plans.

To provide management control, the concept of *baselines* is introduced. A baseline [12] is the foundation of the SCM discipline. It is a defined state which SCIs pass at a specific time during their life-cycle. The establishment of a baseline is generally carried out to indicate that the associated SCIs conform to some requirements or exhibit some characteristics [105]. Before a SCI becomes a baseline, change may be made quickly and informally. However, once a baseline has been established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

According to the STARTS Guide [105], SCM is considered to cover:

- *Version and Variant Control*, which identifies the SCIs and records the history of their evolution through successive versions.
- *Configuration Control*, which is concerned with the building of appropriately structured systems from their constituent parts.
- *Change Control*, the operation of applying changes, to establish new states through which the project passes.

Although the SCM discipline is essential throughout the software life-cycle, it is even

more important during the maintenance phase. Software maintenance is concerned with changing existing software systems; SCM provides precisely the framework that is needed to manage such changes. Indeed, most software systems problems are often more acute during the maintenance phase when the largest number of SCIs, which exist in many versions and which are highly dependent upon each other, must be managed.

2.4.1 The Functions of SCM

SCM does not provide a design method or life-cycle model, nor does it define how the quality of items is to be judged. It does, however, provide a solid foundation for all other software engineering activities. This achievement is accomplished by the four main functions of the discipline which are: identification, control, status accounting and auditing.

Software Configuration Identification Function

Software configuration identification is a process that ensures meaningful and consistent naming for all items in the software configuration. This function exposes the constituent parts of a software representation in a manner which explicitly manifests the relationship among these parts. It is the process by which any piece of software is transformed into a structured entity. This SCM function provides the mechanism for obtaining visibility and establishing traceability of the SCIs which comprise a baseline.

Software Configuration Control Function

This function is designed to respond to the need for change, but it also serves a more general purpose in the effective management of the system evolution. It is a framework by which the software portion of a system can achieve and maintain visibility throughout the life-cycle. It provides the procedures necessary for proposing, evaluating, reviewing, approving and implementing changes to a baseline. Without these procedures, uncontrolled changes might cause more problems than they solve. All changes to a SCI should be controlled by using a formal procedure to obtain authorization to make the change. The body which authorizes change is usually known as a Configuration Control Board (CCB). The CCB must be seen to have the authority to evaluate proposals and authorize the implementation of changes to the software system. It is also a role of the software configuration control function to establish the standards for software development and maintenance, in order to prepare new baselines for review by configuration auditors.

Software Configuration Status Accounting Function

This function provides the mechanism and tools for recording and reporting the current status and evolution of a software system throughout its life-cycle. This SCM function is satisfied when the outputs of the SCM identification, control and auditing functions are recorded, stored and can be reported.

Software Configuration Auditing Function

The purpose of this function is to increase the visibility of the software system and to establish traceability throughout the life-cycle of the software system. A successful auditing should result in a baseline which checks that each to-be-established

baseline possesses the appropriate technical relationship to existing baselines. The two fundamental processes within software configuration auditing are verification and validation [21]. The verification process is largely an administrative function, while the validation process involves a technical assessment of the baseline. *Verification* entails evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase. It ensures that the correct and current versions of all product parts are included in the baseline, that traceability to the previous formal baseline is included, and that the product parts have the correct logical identification. *Validation* entails the evaluation of software at the end of each development effort, to ensure that it meets its initial requirements. The maintenance validation involves regression tests, which help confirm the absence of unanticipated side-effects in functions not related to those being modified. It may also involve auditing software for adherence to design principles, coding standards and other quality standards.

2.4.2 Automation of SCM

The SCM discipline has been widely employed and automated in recent years. The application of SCM throughout software development has shown to be an efficient method of improving the reliability and the quality of software produced. As recently as ten years ago, the discipline of SCM was considered of little value, mainly because it was performed manually. Therefore, project managers could not usually rely on the SCM organization, because its performance was too slow and the work was error-prone.

In recent years, systems have been developed to automate the SCM process. Some

of the first systems, such as SCCS [93] and RCS [108, 109] deal primarily with source code versioning and storage. These systems provide good support for keeping track of versions of files in a software system, but provide only marginal support for understanding the structure of a large software system consisting of many modules, and for keeping track of relations between documents, source code and test cases.

Other tools, such as *make* [39], emphasized the system-building aspects of SCM. However, *make* builds the system using the latest versions of the files, with little regard for the previous versions. Several solutions have been proposed to integrate a version control system with a configuration management system, so that it would be relatively easy to identify and generate a specific version of a system, consisting of many modules. The work described by Shigo *et al* [101] is an early attempt in this direction. The more powerful systems, such as Cedar System Modeller [69], and DSEE [73, 74] also provide the integration of source-versioning and system-building. This integration is essential for the large and complex software systems currently being developed and maintained.

In small projects, the dominant aspect of SCM is the detailed recording of the who, what and when of each change made. To accomplish this, most projects use source-versioning systems. In larger software developments, however, communication and control become the dominant factors in managing change. There must be a set of well-defined procedures for reporting problems with the product, recommending changes or enhancements to the product, ensuring that all parties with an interest in a change are consulted prior to the decision being made to incorporate it, and ensuring that all affected parties are informed of the schedules associated with each change to the product. As a result, automated systems are considering the need to record, track and analyse volumes of data, and to maintain complex

cross-references between large documents. Online systems have greatly reduced the amount of paper necessary. They enable SCM personnel to report accurate and timely information about the status of any requested change to any baseline to the project managers [11].

Ambriola *et al* [2] presents an interesting classification of the evolution of systems for configuration management and version control, from simple stand-alone tools, such as *make* and SCCS, based on an underlying file system towards more integrated systems based on a project database. They have distinguished three generations of the evolution of tools and environments.

The first generation tackled the problem of having two separate tools for performing the related tasks of SCM and version control. Although these tasks are integrated, they are not part of a software development environment. Moreover, these tools were working over unstructured text files. An example this generation's tools is RCS, working together with *make*. RCS, which is primarily a tool for version control, provides a simple interface to *make*, in a naive attempt at integrating the activities of SCM and version control.

In the second generation, tools performing the related tasks of SCM and version control were integrated with the rest of the software development environment. The systems of this generation moved away from the file system point of view, introducing the notion of a database. One weakness presented by systems of this generation derives from the assumption that the employed programming languages have no constructs for intermodular type-checking, and therefore have no means of expressing module interconnection. The consequence of this assumption is that these systems require an external description of each component in terms of its import/export

features and of the other components it uses.

This approach is convenient when different languages are used for different components, but it becomes an overhead in environments supporting languages which actually contain such constructs. This overhead consists of a lot of redundant information which has to be manually provided by the users. Systems classified into this category are the Cedar [69] and Gandalf [51, 59] environments. The Domain Software Engineering Environment (DSEE) [73, 74] is another system which can be categorized as belonging to this generation, and overcomes many of the limitations of SCCS and *make*. The DSEE system, which runs on Apollos, has a rich set of facilities for version management and keeping track of source configurations using configuration threads. The Shape system [79] also integrates SCM with version control, by extending makefiles of *make* to include rules of selecting specific versions of objects. However, it does not provide relationships between specifications, design and code.

The third generation systems are oriented towards high-level languages. These systems take advantage of an underlying database, which provides a rich set of attributes that can be used for choosing the components needed to instantiate a system. Another advantage of having a database as central repository is that the integration among different tools can be made very strong. The Eclipse Environment (with the SCM facilities more extensively covered in its first version) [1, 18], is an example of system of this generation. Adele [7, 37, 38] is another example; it is a database management system for program modules, supporting multiversion software, relationships among software components, configuration management, and access control and protection. The automated software engineering framework developed by the Planning Research Corporation (PRC) [91] can also be categorized

as belonging to this generation. This framework aims to develop and maintain ADA software systems. It supplies the mechanisms to enforce SCM policies, and automates the labour intensive clerical aspects of SCM, such as tracking and maintaining the historical information necessary for SCM status reporting and auditing.

From this brief analysis of the evolution of the application of SCM and version control, it can be seen that SCM and version control should be performed within an environment, so that they can be integrated with the other maintenance activities. A key aspect of this integration would be a database as the central repository of system knowledge and information, so that SCM and version control can be performed with reference to the conceptual schema underlying the environment.

2.4.3 Application of SCM to Software Maintenance

The SCM discipline has been used to improve the software development process with great success. Experience has shown that good management can make the difference between project success and failure. By following sound management practices, development projects can be kept on schedule, resulting in increased reliability, improving the quality of systems produced and increasing job satisfaction. However, these same practices have not been applied to managing maintenance activities of existing software systems.

Traditionally, software maintenance has been treated differently from new development. Because it has been viewed as less difficult and less important, maintenance has been performed by less experienced personnel under less management supervision. The result is that management problems often outweigh the technical

problems of performing software maintenance. Therefore, a management discipline which guides the maintainer through the maintenance process is required.

The SCM discipline which is used to control software development can and must be used to control software maintenance [80], so that the task of maintenance does not cause the software systems to deteriorate after changes are introduced to them, and maintenance becomes more cost effective and reliable.

The automated support for the SCM discipline discussed in this chapter so far mainly deals with its application during software development. As far as software maintenance is concerned, few projects have been related to SCM. The Evolution Support Environment (ESE) [89] provides integrated support for the management of software architecture configuration, life-cycle configuration and version control. Software architecture configuration management allows tracking of interconnections among the software components which make up a system. Life-cycle management allows traceability among specifications, design, code and test cases during software development. Version control allows specific versions of software objects and their associated objects, such as specifications and test cases, to be retrieved.

One project being carried out at the University of Durham deals explicitly with configuration management within the maintenance process. Inverse Software Configuration Management [63, 64] aims at identifying and documenting configurations of existing systems in order to bring these software systems under configuration control. Therefore, this project is concerned with the configuration identification function, and performs the first step aimed at regaining control over an existing system. Proforma Identification Scheme for Configurations of Existing Systems (PISCES) is a tool under development, designed to help in this area. However, to improve the

reliability of an existing system during its maintenance, traceability and consistency among the set of documents and source code should not only be established, but should also be preserved when changes to the code are made.

2.5 Summary

This chapter lays a foundation for the research carried out in this thesis by giving an overview of the concepts of the software maintenance process, documentation for software maintenance and software configuration management. Additionally, this chapter has briefly surveyed a number of related automated support environments, and has outlined their oversights and weaknesses.

The discussions in the previous sections have shown inadequacies in the current ways of tackling software maintenance. Based on the survey, it is evident that further research on software maintenance is required in order to overcome the deficiencies and limitations of existing software maintenance techniques. Good management of software maintenance is an achievable objective, but there is inadequate technical support from methods and tools. Many tools in current use derive from the initial development phase, and are not well suited to the needs of maintainers. Work is needed on methods in order to solve the problems of configuration control and release, and for developing maintenance methods for existing software systems with associated quality assurance procedures.

The prominent software maintenance environments available are taking the approach of reverse-engineering existing software systems, aiming at applying modern software

development techniques to control their evolution. In this sense, the work proposed in this thesis differs from them. The proposed method aims to provide an alternative solution for software maintenance problems by incrementally recovering the documentation while maintaining the software systems. Incremental documentation, as discussed in this chapter, seems to be a viable strategy, concentrating only on those parts of the system which require modification, leaving the remainder alone. Furthermore, the software maintenance process is performed under the control of the SCM discipline. Hitherto, there have been no projects which tackle software maintenance from this point of view.

Some similarities with the arguments posited by this thesis, as far as providing support for maintenance is concerned, can be found in the ASU software maintenance environment [33] and ESE [89]. However, the ASU environment does not deal with SCM control in the maintenance process. Although ESE applies SCM to control the process, this environment does not deal specifically with the maintenance process; therefore, it lacks the support necessary to import the existing software systems into the environment.

The similarity between the approach proposed by this work and large projects such as REDO, MACS and ReForm is that the proposed method (COMFORM) also aims to provide a method for software maintenance and places considerable emphasis on a single integrated representation of the original system in an object base. These environments, however, aim to reverse engineer the software systems and hand them over to current software engineering techniques in order to carry on their maintenance.

The novelty of the COMFORM method is that, unlike the approach to software

maintenance taken by these large projects, it takes a less drastic approach to the incrementally-recovered documentation of existing software systems while the maintenance process is being performed. Therefore, COMFORM is not only a reverse-engineering method. It tackles the problem of incremental documentation, and at the same time, it provides guidance to carry on maintenance in a controlled way by applying SCM. Consequently, control over software systems is improved as maintenance is performed. The benefit of this technique is that it is a less expensive method, by which the maintainability and documentation of software systems increase as they are maintained. Moreover, the method can be routinely used to control the inevitable process of change.

The next chapter discusses existing software maintenance models, and introduces the characteristics of the modelling developed in this thesis. The aspects of creating a single integrated representation of the original system is expected to achieve by formalizing the method. This is obtained through the modelling of the method using ORM discussed in Chapter 4.

Chapter 3

Software Maintenance Models

As the size and complexity of software systems grows, so too will the maintenance burden unless active measures are taken to plan for it and migration to software environments designed to minimize maintenance occurs. Imposing a structure onto the maintenance process of existing software systems may reduce the difficulty of the whole task by refining it to a number of tasks of reduced complexity. In order to structure the maintenance process, it is necessary to model it as in the software development process.

In this chapter, several models for improving software maintenance are analysed, along with their strengths and weaknesses. Additionally, desirable capabilities of software process modelling are discussed in order to introduce the characteristics of software maintenance modelling.

3.1 Managing the Software Life-Cycle

Because of the intangible nature of software systems, effective management relies on adopting models which make the software process visible by means of documents, reports and reviews [104]. This has resulted in the adoption of software models where the software process is split into a number of phases and each phase is deemed to be complete when some deliverable document has been produced, reviewed and accepted.

For sometime, the software process has played a major role in the field of software engineering [78]. During this period, the study of software processes has led to the development of various life-cycle approaches that can be employed in engineering software. The basic function of a life-cycle model in the development of a software system is to describe the chain of events required to create and maintain a particular software product.

The waterfall model is by far the most widely adopted software life-cycle model [104]. This model is typically high-level and does not address detailed activities of the software life-cycle. Although it suffers from inadequacies, it continues to be widely used as it provides benefits as summarized by Sommerville [104]: "it simplifies management of the software process", and by Pressman [88]: "it is significantly better than a haphazard approach to software development". Among the benefits of life-cycle models are the facts that they help us to become aware of, and gain an increased understanding of, the software process, and to determine the order of global activities involved in the production of software. These benefits may result in improved product quality, increased effectiveness of methods and tools, reduced software development and maintenance costs, and increased customer and developer

satisfaction [78].

Despite some variations, the main phases of a traditional software life-cycle model are: analysis (requirements and specifications), design, implementation and maintenance, with possible feedback loops between phases. The current view of the software life-cycle, however, mainly emphasizes the development phases. It does not portray the system life, i.e., it does not show the evolutionary development which is characteristic of most software systems, only showing the creation and development of a software system. The traditional software life-cycle model has always shown the software maintenance activity as a single phase at the end of the cycle. This final phase needs to be replaced by a model which reflects this aspect of software evolution [9]. What is required is a new, broader perspective of the software life-cycle, emphasizing change, maintenance and migration to new technologies. On the basis of the discussion so far, this final phase should also break the work to be done into tasks or stages which can be performed in such a way that the quality cycle concept can be used to control the quality of the software maintenance stages and the individual tasks; progress can then be reviewed at the end of each stage.

3.2 Existing Software Maintenance Models

It is now quite common to divide the software development process into separate phases. Similar models have been proposed for software maintenance. A maintenance model describes the individual steps necessary to satisfy an individual maintenance request. Different types of maintenance requests, environments characteristics, and/or budgetary constraints may require different models [94].

One of the causes of many maintenance problems is the lack of good maintenance models. Practical software maintenance models tend to be more *ad hoc* and are performed without easy access to the necessary information. In addition, the maintenance technology level (i.e., methods for design and code reading, and automated tools) is much lower than during development. The result of all the above is unproductive and error-prone maintenance.

A number of authors have proposed models for software maintenance. The software maintenance models described in this section outline basic guidelines (phases) which, the authors suggest, are to be followed during the maintenance of software systems. These models have been proposed in the last two decades and have slightly evolved to keep up with new technologies that have emerged. These models have been divided into three separate sections to show their evolution, reflecting the chronological order in which they were developed. Accordingly, the first models are typically high level, giving only general guidance, and not addressing detailed activities of the maintenance process. The recent ones, though also high-level, introduce more advanced software engineering concepts and ideas.

3.2.1 Early Software Maintenance Models

The early models of software maintenance were very simplistic. They provided the order of phases to be followed, without addressing the details of how they should be performed.

Boehm Model

Boehm [16] outlines three major phases of a maintenance effort in his model. However, he does not detail the exact maintenance tasks within each phase. The three phases of the Boehm model are:

1. *Understanding the existing software.*
2. *Modifying the existing software.*
3. *Revalidating the modified software.*

Liu Model

The significance of the existing software system is highlighted in the model devised by Liu [76]. Unfortunately, he does not provide any details regarding the tasks within each of the phases of his model. The Liu model consists of a high-level, general model which comprises three phases:

1. *Understanding* of the capacity, function and logic of the existing software system.
2. *Designing of new logic* to reflect the new request or additional feature.
3. *Merging new logic* with existing logic so that the new logic is integrated into the existing software system.

Liu puts forward some suggestions to improve the documentation of software systems and stresses its importance for software maintenance. The documentation, however,

is not part of any of the phases of his model. Liu also emphasizes that strict testing procedures should be followed, although a testing phase is not itemized in his model.

Sharpley Model

The Sharpley model [100] focuses more specifically on the process of correcting errors in existing software systems (corrective maintenance). The phases of the Sharpley model are:

1. *Problem verification.* Identify cause, reproduce trouble situations and verify reported symptoms.
2. *Problem diagnosis.* Isolate the software components which are in error, evaluate problem severity and estimate cost to fix.
3. *Re-programming.* Modify code and generate a new system version.
4. *Baseline verification/re-verification.* Ensure the correctness of the modified software system.

Sharpley emphasizes that all phases of the software maintenance process involve some sort of testing, which consists of verifying key states and major transitions on a selective basis.

Yau and Collofello Model

Yau and Collofello [115] identified four basic phases in managing the software maintenance process. Their model focuses on software stability through analysis of the ripple effect of software changes. These activities can be accomplished in the four following phases:

1. *Understand the program.* Consists of analysing the program in order to understand it.

2. *Generate a maintenance proposal.* Consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified.

3. *Account for ripple effect.* Consists of accounting for all of the ripple effects as a consequence of program modifications.

4. *Testing.* Consists of testing the modified program to ensure that it has at least the same reliability level as before.

Each of the four phases are associated with software quality attributes. The first phase is associated with complexity, documentation and self-descriptiveness attributes, which contribute to the easy the understanding of a program. In the second phase, the ease of generating maintenance proposals for a program is affected by the attribute extensibility. Yau and Collofello state that one of the most important quality attributes is the stability of the program (in phase three), because if the stability of a program is poor, the impact of any modification on the program is large. In the fourth phase, the primary factor contributing to the development of cost-effective testing techniques is the testability of the program.

3.2.2 Middle Software Maintenance Models

The models described in this subsection are more elaborate than those in the previous one. Martin and McClure's model, for instance, provides further details on how to perform the task of maintenance; while Patkau's model provides detailed functions to be performed according to the maintenance category. Arthur's model is more comprehensive, providing guidelines for the change request to system release phases.

Martin and McClure Model

The high-level breakdown of the tasks offered by the Martin and McClure model is similar to that offered by other investigators. The three basic functions are:

1. *Understand program.* This entails understanding the functional objective, internal structure and operational requirements of a program. This function is subdivided in three subfunctions:

    - Top-down understanding. A top-down approach should be used to become familiar with a program at a general level of understanding, then at detail level.
    - Improve documentation. During the understanding of the program, document what is learnt, concentrating on improving high-level program documentation.
    - Development participation. When possible, participate in the program development process to learn about the program.

2. *Modify program.* This involves creating new program logic to correct an error

or to implement a change, incorporating that logic into the existing program. It necessitates the following steps:

- Devise a plan for changing the program. A top-down approach is recommended to review the program. First, the modules and data structures to be changed are isolated. Next, the internals of each module and data structure to be changed are studied in detail. The change is then designed, specifying the new logic and any existing logic that must be altered.
- Alter the program code to incorporate the change. The objective of this step is twofold: to correctly and efficiently code the change, and to eliminate any unwanted side effects from the change.

3. *Revalidate program.* The maintainer should perform selective retesting to demonstrate that not only is the new logic correct, but also that the unmodified portions of the program remain intact and the program as a whole still functions correctly. The maintainer should:

   - Test for program failure by performing system tests to be certain that the entire system is still operating correctly as a whole.
   - Test the unmodified portions of the program by performing regression tests, to determine if those parts still operate correctly.
   - Test the modified portions of the program to determine if the changes were designed and implemented correctly.

Patkau Model

A more comprehensive approach to software maintenance is presented by Patkau [82].

Patkau first presents a generalized high-level model which identifies five basic maintenance phases. His model is then further refined, and four versions of the generalized model are presented. Each of these versions models the four categories of software maintenance: corrective, perfective, adaptive and preventive maintenance. An important feature of this model is its emphasis on specification and localization of the change. The five phases of the generalized model are:

1. *Identification and specification of the maintenance requirements.*
2. *Diagnosis and change localization.*
3. *Design of the modification.*
4. *Implementation of the modification.*
5. *Validation of the new system.*

The refined versions of the generalized model differ in the first two phases. The remaining phases of the model are similar among each of the four categories of software maintenance. Although Patkau provides a further refinement of these phases, they will not be discussed further as they are similar to those of previous models.

For perfective maintenance, the two initial phases are:

- Identify new or altered requirements, and specify operation of the enhanced system.
- Locate affected elements.

For adaptive maintenance, the two phases are:

- Identify a change in the environment. Describe the change and revise specifications to reflect it.
- Locate the elements affected by the change.

For corrective maintenance, the first two phases are described as:

- Identify repeatable error symptoms and specify correct operation of the system.
- Locate the part of the system responsible for the error.

For preventive maintenance, the two first phases are refined to:

- Identify a deficiency in performance, maintainability, etc., and specify desired performance or quality standard.
- Locate the source of the deficiency.

Arthur Model

Arthur [4] proposes a more elaborate and comprehensive model for software maintenance. His model presents phases to deal with the request for changes until they are implemented, tested and released to the users. The seven phases of his approach to tackling the software maintenance process are:

1. *Managing change.* The basic objective of change management is to identify, describe and track the status of each requested change. In this phase, change requests are generated and analysed.

2. *Analysing change.* The overall objective of impact analysis is to determine the scope of the requested change as a basis for planning and implementing it. Change requests are evaluated for potential impact on existing systems, other systems, documentation, hardware, data structures and humans (users, maintainers, and operators). A preliminary resource estimate is developed.

3. *Planning system releases.* The principle objective of system release planning is to determine the contents and timing of system releases. Change requests are ranked and selected for the next release. Changes are batched by work product, and the work is scheduled.

4. *Designing changes.* The major objective of the design phase is to develop a revised logical and physical design for the approved changes. Logical design relates to the system level, and physical design relates to the program level.

5. *Coding changes.* The objective of coding is to change the software to reflect the approved changes represented in the system (logical) and program (physical) designs.

6. *Testing changes.* The primary objective of testing is to ensure compliance with the original requirements and the approved changes.

7. *Releasing the system.* The objective of system release is to deliver the software system and update documentation to users for installation and operation.

3.2.3 Recent Software Maintenance Models

The most recent models dedicated to software maintenance are illustrated in this section by the Foster, Pfleeger and request-driven models. While Foster's model

tries to approach maintenance from the organizational viewpoint, Pfleeger's model is an attempt to improve the maintenance process by managing it through metrics. The request-driven model attempts to portray the activities of software maintenance as dictated by user requests for change involving strict control from management.

Foster Model

Foster *et al* [42] argue that the organization of software maintenance is of critical importance to the success of the activity itself. Their model is derived from the observations made by actual maintenance teams and covers technical and managerial issues. This model differs from others discussed in this chapter in that it does not dictate the phases to be followed during the maintenance process, but refers to the functions (duties) performed by people involved.

The model specifies the four following functions which are triggered by queries, problem reports and change requests sent by customers.

1. *Front desk* receives these communications from customers and retains records of them. It is the responsibility of the front desk to provide answers/solutions, either directly or by passing on the request to a more specialized duty within the team.

2. *Request store* receives the requests which require new solutions. The request is queued until effort becomes available. This queue is represented as the request store, which contains a backlog of unactioned requests. The management of the request store is an important function. Priorities are assigned among its contents, and preliminary investigations and impact analyses are performed in order to plan future work. If the preliminary investigations reveal that

the team does not have the resources or capability to provide the answer to a problem, a request must be made to some other team for a solution to be provided. If that team can cope with the request, it will be dealt with as one of a repeated series of actions, in which the highest priority request is taken from the store and the software change designed.

3. *Change store* accumulates changes and solutions received from other teams. From time to time, the decision is taken to build a new release of the software, incorporating all new changes available.

4. *Solution store* provides a set of known answers/solutions which are accessed by the front desk. This represents information in a variety of forms, such as versions of software products and paper records of answers to frequently asked questions. If the solution is in the store or can be quickly generated, then it is immediately solved and issued back to the customer. New solutions are also lodged in the solution store, from where they are available for distribution to the original customers.

Pfleeger Model

Pfleeger [86] describes a model for software maintenance which emphasizes impact analysis and forms a framework for software maintenance metric support. The model incorporates metrics for assessing and controlling change. Pfleeger's model can be seen as an improvement on that of Yau and Collofello, as the software quality attributes are associated to the phases so as to monitor product quality. The major activities of this model are:

1. *Manage software maintenance.* This controls the sequence of activities by

receiving feedback with metrics and determining the next appropriate action.

2. *Analyse software change impact.* It evaluates the effects of a proposed change. If the impact of change is too large, or if traceability is severely affected by the change, management may choose not to implement the change.

3. *Understand software under change.* Source code and related product analysis are needed to understand the software system and the proposed change. The likely degradation of system characteristics, such as complexity of the system, self-descriptiveness of the source code and documentation quality helps to decide if the change will be implemented or not.

4. *Implement maintenance change.* This generates the proposed change. Adaptability of the system is analysed to perceive the difficulty of implementing the change.

5. *Account for ripple effect.* This analyses the propagation of changes to other code modules as a result of the change just implemented. Stability, coupling and cohesion of affected modules serve to check the original impact analysis effectiveness.

6. *Retest affected software.* The modifications are tested to meet new requirements, and the overall system is subject to regression testing to meet existing ones. Testability, completeness and verifiability are observed in this activity.

Request-driven Model

The request-driven model [9] attempts to portray the activities of software maintenance as dictated by user requests for change. The model consists of the following three major processes, which involve strict control from management:

1. *Request control.*
2. *Change control.*
3. *Release control.*

The *request control* process deals with the user's requests for change. The major activities which take place during this process are:

- Collection of information about each request.
- Setting up of mechanisms to categorize requests.
- Use of impact analysis to evaluate each request in terms of cost/benefit.
- Assignment of a priority to each request.

Change control is often seen as a key process, the most expensive activity being the analysis of the existing code. The activities involved in this process are:

- Selection of changes from top of priority list.
- Reproduction of the problem (if any).
- Analysis of code, documentation and specification.
- Design of changes and tests.
- Quality assurance.

It is during the *release control* process that requests which are to be included in a new release of the software system are decided and the necessary changes to the source code made. The activities which take place during this process are:

- Release determination.
- Building of a new release by editing source, archival and configuration management, and quality assurance.
- Confidence testing.
- Distribution.
- Acceptance testing.

The software maintenance models discussed in this section provide a means of communication between the people involved in the maintenance task, assistance in the management, and some may provide a foundation for building maintenance-driven tools. However, because these models provide mainly high-level guidelines, they do not present sufficient details of actions and events necessary to ensure the quality and maintainability of software systems which should precede each new release. Existing maintenance models lack a framework with rigid guidelines, in order to make certain that software systems are improved rather than impaired after each change is introduced.

3.3 Software Process Modelling

Models for software development, as is the case for most of the software maintenance models described in the previous section, represent the software process in terms of phases. Additionally, software maintenance models only characterize the maintenance process from the maintainer's perspective. The structuring of maintenance activities provides a useful mechanism for improving the process. However,

the application of these models has been of limited benefit in actually aiding the maintenance process. Moreover, existing models do not describe the actual processes which occur during software maintenance; they may only provide a means of visualizing the process in terms of interim products. Phase milestones could be associated with these products, thus providing a mechanism by which management satisfaction could be assessed with respect to general requirements, budgets and schedule.

Moreover, software maintenance models, like software development models, should provide wider aid to fully encompass the whole process. Kellner [62] defines software process modelling as a methodology that encompasses a representation approach, comprehensive analysis capabilities and the ability to make predictions regarding the effects of changes to a process.

Several diverse goals and objectives have been cited as motivation for the development and application of software process models. These include support for automated execution and control, human interaction (such as execution guidance), various management responsibilities, process understanding and analysis of processes [61].

Gallagher [45] defines a software maintenance process model as the specification of a systematic approach to the maintenance of software. With the purpose of extending the experience and technology of software development process to software maintenance, the four primary objectives for the development of software process models, described in [62], should be taken into account:

1. Enable effective communications, regarding the process to others (workers,

managers and customers).

2. Facilitate reuse of the process by enabling a specific software process to be instantiated and executed in a reliably repetitive fashion across multiple software projects.

3. Support evolution of the process by serving as a repository for modifications, lessons learnt, and tailoring, and by analysing the effectiveness of changes in a laboratory of simulated environments before actually implementing them. Successful tailoring decisions should then be formalized and stored as part of the model, so that they can be consistently applied in the future.

4. Facilitate effective planning, control and operational management of the process. This is accomplished through increased understanding, training, conformity to process definitions, quantitative simulation and analysis capabilities, and definitions and use of measurements and metrics.

In order to accomplish these objectives, software process models must possess capabilities in three major categories:

- A powerful representation formalism is required to cope with the complexities of actual organizational processes.
- Comprehensive analysis capabilities, including a wide variety of tests in the areas of consistency, completeness and correctness. They are critical in determining the validity of the model itself, and of the actual process the model represents.
- Forecasting capabilities which can be provided through simulation that is tightly integrated with the model representation and analysis features.

From these observations, Kellner and Hansen [62] summarize the requirements for an ideal approach to software process modelling. Similar lists have been presented elsewhere. For example, a summary of the capabilities discussed at the *4th International Software Process Workshop* [58] is presented in [92]. Rombach and Mark [95] also list numerous desired capabilities. These characteristics are described below.

1. Use a highly visual approach to information representation, such as diagrams.
2. Enable compendious descriptions, i.e., comprehensive in scope, yet concise in presentation.
3. Support multiple, complementary perspectives of a process, such as functional, behavioural, organizational and conceptual data modelling.
4. Support multiple levels of abstraction (e.g. hierarchical decomposition) for each perspective.
5. Offer a formally defined syntax and semantics, so that the constructs are computable.
6. Provide comprehensive analysis capabilities. This would involve tests in categories such as consistency, completeness and correctness.
7. Facilitate the simulation of process behaviour directly from the representation.
8. Support the creation and management of variants, versions and reusable components of process models.
9. Support the representation and analysis of constraints on the process, such as regulations, standards and so on.
10. Enable the representation of purposes, goals, rationales, and so forth, for process components and the overall process.

11. Integrate easily with other approaches which may be deemed useful.

12. Take an active role in process execution.

13. Offer automated tools supporting the approach.

Techniques for software process modelling are still under development and the availability of even a portion of the requirements described above is expected to bring substantial benefits to the software development and maintenance processes. Besides, these requirements may in the future, facilitate the evolution of software processes in a methodical and disciplined fashion.

3.4 Characteristics of Software Maintenance Modelling

Hinley and Bennett [54] argue that process models need to have the following characteristics in order to provide real benefits for maintenance management:

- A comprehensive coverage of the software maintenance process, but avoiding complexity.
- A large scope addressing organizational, behavioural and functional aspects.
- Recognition of real-world objects by the model (change requests, fault log).
- Recognition of explicit roles of the people (manager, customer, user) who interface with the maintenance process.

- A suitable diagrammatic form so that measurement and control points can be established.

- Recognition of the process communication pathways, which may not reflect an activity sequence or organizational hierarchy, e.g. management control mechanisms.

- A framework which guides managers in their use; for instance, how they can measure process performance against stated goals, with the aid of a model.

- Reusable features i.e., provide process modules or templates which can be refined or tailored to suit individual maintenance project circumstances, such as the change request control process and the change release process.

- Flexibility, so that models can be quickly adapted to cater for real process transformations and changing relationships.

The requirements for software process modelling, in addition to the characteristics for maintenance management described above, have been analysed in order to provide a basis for comparison with the proposed software maintenance modelling.

Although the modelling supporting the proposed method is simple, it provides most of the desirable capabilities necessary to generate a supportive maintenance process. In addition, it provides aid for maintenance management, as detailed below;

- It covers each maintenance activity from change request to its release to users.

- It addresses a number of different aspects of the maintenance process, such as functional and conceptual data modelling aspects.

- The objects of the formalization are real-world elements such as change proposal, maintenance specification and configuration release.
- It considers the people involved in the maintenance process and their roles.
- The functional process model provides a diagrammatic form of representation which allows measurement and control points to be established.

A relevant aid for the management of the maintenance process is the functional process model, which defines the framework necessary to systematize the software maintenance process, by specifying the chain of events and the order of stages that a change has to go through. This model:

- Comprises ways of coping with the need and justification for changes.
- Improves the flow of maintenance activities by providing guidance throughout the maintenance process.
- Provides a formal change control procedure to monitor changes and protect software quality.
- Determines the organization and content of the information needed to support the maintenance activities.

3.5 Summary

In this chapter, a survey of software maintenance models has been presented. Although there are a relatively large number of software maintenance models, they

have displayed oversights which require further research into this field. Relevant characteristics of software maintenance modelling have also been discussed, in order to introduce the necessity of generating alternative approaches to improve the maintenance of existing software systems.

This research intends to develop a software maintenance method, aiming to provide a consistent and structured approach to software maintenance, overcoming disastrous documentation, reducing dependencies on experts and delivering productivity in maintenance. Some characteristics of software process modelling are employed in order to provide the necessary guidelines for the proposed method. Chapter 5 provides details of the functional software maintenance model proposed in this thesis. The next chapter explains the background of data modelling techniques so as to introduce the conceptual data modelling aspects of the proposed method.

Chapter 4

Modelling Techniques

Models, in general, are abstractions designed to understand a problem before implementing a solution. Because a model omits non-essential details, it is easier to manipulate than the original entity. To build a complex system, the developer should abstract different views of the system, build models using precise notations, verify that the models satisfy the requirements of the system, and gradually add detail to transform the models into an implementation. Moreover, models are useful for communicating with application experts, modelling enterprises, preparing documentation and designing systems and databases.

The approach to modelling and formalizing the software development process has recently been receiving attention, since such a strategy provides great benefits to clear expression of abstract concepts. Additionally, it may provide both an insight into the development process and the necessary means to design more supportive

software engineering environments.

Effective configuration management requires the application of knowledge from the underlying software process, which can adequately represent software and express semantics capabilities [96]. Therefore, software modelling and formalization are important steps in applying and automating the SCM discipline in a software environment.

This chapter provides a description of various approaches to model software systems. In the following sections, traditional modelling techniques, such as the relational, entity-relationship and object-oriented models, are reviewed along with their characteristics and applications to model software systems. These data models are presented so as to introduce the Object Representation Model (ORM), and to put it in the context of data models. The ORM, with its main characteristics, is then described, along with its contribution to the modelling of the method proposed in this thesis.

4.1 The Relational Data Model

In the 1970s, the relational data model [32] was the focus of much research in the database area; it spawned considerable theoretical, as well as implementation activity. Relational database technology is now well understood, and a number of relational database systems are commercially available; they support the majority of business applications relatively well. The relational model is accepted as the state-of-the-art model in the commercial database field.

A relational database consists of a collection of relations, or tables, and a specification of underlying domains for the entries in the table. The table entries are atomic values. Each table may be viewed as having a fixed number of columns and a variable number of rows, the order of which is unimportant. Data in several tables is related through matching column values, rather than through the specification of explicit links. A major advantage of the relational model is that it has a formal mathematical basis, whereas prior to its introduction, the entire database field could be viewed as a large collection of *ad hoc* methodologies. In particular, the relational theory provided a formal basis for assessing the quality of a specific database design.

However, it has been recognized that this simple conceptual data model, however elegant, is of limited use in modelling applications, where the data is confined to a small number of different types, related in well-defined ways. This restriction may not have been a problem in many of the applications for which it was originally intended, such as banking transactions and inventory management. Nevertheless, its usefulness has been limited when applied to complex, highly structured application domains, such as some recent application areas in need of database management support, like Software Engineering Environments (SEE). These applications place demands on database systems which exceed the capabilities of relational systems. In other words, the basic relational data model has been proven incapable of capturing and controlling much of the semantics of complex applications with such a simple framework.

The deficiency of expressive power in the relational data model leads to semantic overloading, when a single data model construct represents different abstract concepts. For example, the primitive data construct relation (table) is used to represent both objects and relationships. Moreover, the simple record structure, where

attributes are atomic, has also been considered as a major disadvantage of relational systems, because it is difficult to express the semantics of complex objects in this fashion.

Given the inadequacies of the relational model, new data models have been defined, often as extensions of the relational data model, which are more expressive and can capture more of the semantics of the application domain. Such models are termed semantic data models.

4.2 Semantic Data Models

The emergence of semantic data models was mainly due to the necessity of mechanisms for improving the representation of data. Semantic data models usually include a rich set of structural abstractions which can be used by the data modeller [68]. In this way, semantic data models may provide facilities for differentiating between kinds of data entity, related through various kinds of relationships. By distinguishing the many kinds of entity and relationship types which can exist, these models can be more expressive when representing the semantics of the application domain.

4.2.1 The Entity-Relationship Model

The Entity-Relationship (E-R) model [28] is the most well-known semantic data model. It was originally conceived of to support the data representation of software

systems which were unable to have the necessary semantic capabilities represented by previous models.

The design representation scheme of the E-R approach contains three classes of elements: entities, relationships and attributes. Entities are elements which can be uniquely identified. Groups of entities may constitute an entity type, such as employee or project. Relationships are conceptual links which exist between or among entities. Relationships can also be classified into different types, such as marriage or project-employee. Attributes are properties possessed by entities or relationships, and have corresponding values. For example, age is an attribute or property of all employees. Relationships may also have values. For example, the relationship marriage has the attribute anniversary date.

The popularity of the E-R model is due to its economy of concepts and the widespread belief in entities and relationships as natural modelling concepts. Nevertheless, it should be noted that in non-conventional applications (where this model tends to be used), modifications or extensions to the original model are usually made to increase its representational capabilities. Several extensions have been proposed to overcome the deficiencies of the E-R model. Accordingly, the E-R model can be considered as an axis, around and on which new implementations add extensions or impose restrictions. A large number of examples of models with extensions, which adopted the E-R model to represent software, can be found in the literature; for example: GENESIS [90], ALMA [70] and VIPEG [66]. More directly related to model configuration management aspects of software are: DMM [77], PACT CM [102] and CMA [87].

Although semantic data models, and in particular the E-R model, have proved to

be excellent ways of modelling data, notably absent from these models is a better support for manipulation of data [68]. That is, their extended data structuring mechanisms are usually accompanied by the same general set of operators (i.e. create entity, delete entity and update entity) [20]. Data would be better accessed and updated by a fixed set of data-type specific operators. Hence, the next stage in semantic data modelling is the integration of operation definition with the data structuring facilities, so that operator definitions are entity-type specific. The object-oriented paradigm has been seen as one way to attempt this integration, providing a mechanism for progressing from a purely structural model of data towards a more behavioural model, and combining facilities for both the representation and manipulation of data within the same model.

4.3 Object-Oriented Models

In object-oriented models, the entities of interest are called *objects*. Much confusion surrounds the definition of the object-oriented models because objects emerged as programming concepts and were therefore driven by implementation considerations [112]. The increasing interest in object-oriented approaches in the last few years has led to a proliferation of definitions and interpretations of this much-employed term.

King [68], in an attempt to define the object-oriented model, compares object-oriented and semantic data models. Semantic data models provide constructors for creating complex types, whereas behavioural issues are often left undefined. In contrast, object-oriented models take an abstract data-type approach to gathering

together operations and data-types. In this way, a data-type is dependent on its own behaviour. Since there is much confusion regarding the definition of what is *object-oriented* in the literature, the distinction between the two sorts of modelling is not always well defined.

Objects of object-oriented models are uniquely identifiable, may have a number of data properties associated with them, which record the current state of the object, and can be manipulated through a well-defined set of operations (or methods). In object-oriented systems, a distinction is made between an object's identity and its properties. As a way of organizing objects in a more manageable way, objects which have the same kinds of properties, and can be manipulated using similar operations, are classified to form a class. Hence, each object is said to be an instance of a *class*. A class acts as a template for creating instances of that class (objects). The objects in a class share a common semantic purpose, above and beyond the requirements of common data properties and a set of operations.

Because of the considerable disagreement concerning the definition of *object-oriented*, the elements of object-oriented models vary from author to author. For instance, Booch [17] considers abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistence as the important elements of an object-oriented approach. Khoshafian and Abnous [67], however, define object-orientation as abstract data-types added to inheritance and object identity. Further, Rumbaugh *et al.* [97] identify the following four characteristics as part of an object-oriented approach: identity, classification, polymorphism and inheritance. Despite having no generally accepted terms and definitions, the most common elements usually referred to and associated with object-orientation are as follows:

- **Identity** means that data is encompassed into a discrete, distinguishable entity called an object. Each object is uniquely identified by an object identifier. The identity of an object has an existence independent of its value. The notion of an object identifier is different from the concept of key in the relational data model. A key is defined by the value of one or more attributes and can therefore undergo modifications; but two objects are different if they have different object identifiers, even if all their attributes have the same values.

- **Abstraction** denotes the essential characteristics of an object which distinguish it from all other kinds of objects, thus providing crisply defined conceptual boundaries, relative to the perspective of the viewer. *Abstract data-types* describe a set of objects with the same representation and behaviour. In most object-oriented programming languages, classes represent the implementation of abstract data-types.

- **Encapsulation** is the process of hiding all of the details of an object which do not contribute to its essential characteristics. It is usually the formal term which describes the bundling of data properties and operations (methods) together within an object, so that access to data is permitted only through the object's own operations. In this way, the concept of object is more closely in tune with the way real-world entities behave; i.e., the concept of what an entity represents is rarely separated from what can be done with it in actuality.

- **Inheritance** is the most important mechanism in arranging classes into hierarchies. It defines a relationship among classes, allowing a class to inherit the representation and behaviour from existing classes. Inheriting representation enables data structure sharing among objects. Inheriting behaviour enables code-sharing (and hence reusability) among software modules. The combi-

nation of these two inheritance factors provides a very powerful strategy for software modelling, development and reuse. Inheritance is achieved by specializing existing classes. Classes can be specialized by extending or restricting their representation (data structures) and/or behaviour (operations) to classes down the hierarchy.

Examples of models which adopted the object-oriented approach to represent software are not easy to find, since this concept is closely related to the implementation aspects of software. Another reason is that the object-oriented paradigm does not provide a "standard" graphical representation for models. As a result, it is usually easier to find conceptual modelling for the representation of data using the E-R model, extended with some concepts of object-orientation, the resulting implementation using an object-oriented language or approach.

Some authors explicitly claim to be using the object-oriented paradigm for modelling software [3]. Others specify that their modelling is E-R model-like, extended with operations and active data to provide the behavioural aspects of the modelling [85]; or that the static aspects are defined by a model based on the E-R model, with an additional formalism to provide the modelling of behavioural aspects of software [6].

4.4 The Object Representation Model

This section describes the Object Representation Model (ORM) [111] formalism. Although ORM provides further capabilities, this section emphasizes only those features which are employed in this research.

Some of the fundamental concepts of ORM have the same features as semantic models. For example, real-world aspects are modelled through objects (entities in semantic models), which may, in turn, be associated with other objects by relationships. Both objects and relationships may have attributes, which have an associated characteristic. Every element of ORM can be classified according to pre-defined standards, i.e., the *types* of the elements. The concept of type is equivalent to the set of entities and relationships of semantic models. However, this concept is also the same as that of object-oriented models which assign a type to each object, aiming at characterizing the properties and operations of the objects.

Concepts within ORM, such as individuality of objects, comply with the object-oriented approach; i.e., objects are handled individually. Therefore, access to a set of objects is a particular case, where the set should be previously identified as an object. Moreover, in ORM every object has a type and comprises an identifier, which can be its principal name defined by the user, or a unique internal identifier, assigned automatically.

The conceptual support of ORM consists of modelling data through objects and information associated with objects. This information may be the specification of some object characteristic (attribute), or the indication of a possible link with other objects (relationship); the latter may also have several attributes.

The mathematical basis for the model is the graph theory. Therefore in ORM, the nodes of a digraph represent the objects of the real world; the arcs represent the associations (relationships) which exist among them.

In general, object-oriented models name all objects of the same *type* as a *class* and the occurrence of each object as an *object instance*. Associations of objects

are made through stored properties of each class, and the association is seldom explicitly stated. In ORM, associations of objects are made explicitly and, therefore, a clarification of terms to designate objects and relationships must be provided. Consequently, the term *object* is used to designate an object instance, and the term *relationship* is used to designate a unique relationship between two given objects. Analogously, the term *object type* is used to designate all objects having the same properties; *relationship type* is used to designate all relationships having the same properties and linking the same object types.

4.4.1 The Cluster Mechanism Concept

The cluster mechanism is an important concept within ORM. It aims to provide a structure for organizing object types. The description of an object can be refined through a set of other objects. Each object of a set depends on (in terms of existence and identification), and is constrained by the object which refines them. The set of objects constrained by an object is termed a *cluster of objects.* A cluster of objects also owns a type, and therefore, objects of a particular type belong to the same particular *cluster type.* In other words, the set of cluster types establishes a partition on the set of object types, in as much as the set of all clusters of objects establishes a partition on the set of objects of an object base.

In this way, similar to other terms in ORM, *cluster type* is used to designate the constraint on a set of object types; *cluster of objects* is used to designate an instance of a cluster type. A cluster type may have several object types associated with it as the designer establishes that they represent a particular focus of interest and thus should be manipulated as a particular set. This concept, however, provides not

only the constraint on a set of objects, but also the mechanism which facilitates the manipulation of such sets for generation of versions within an object base.

Since a cluster type may comprise of other cluster types, they can be organized hierarchically so that detailed system information can be obtained as far down as one goes into the cluster hierarchy. One cluster of objects is permanently available and is named the *global* cluster. The objects of the global cluster are always accessible. In addition, the global cluster type is the most generic one, and comprises all other cluster types which are modelled in a particular software system. Therefore, all system users must have access to it in order to enable other system cluster types to be accessed.

Initially, from the global viewpoint few information of an object base can be accessed. When interest is focused on one of the *global* sub-clusters, only details of this sub-cluster (either objects or other clusters) can be obtained, keeping the same level of details for the remaining system information. Whenever other available clusters of objects become accessible, the level of system detail increases. However, not all objects of the same level become available, but only those which are made up by the cluster of objects chosen to be accessible.

For instance, consider the modelling of the Structured System Analysis methodology [46] using ORM, the cluster hierarchy of which is depicted in Figure 4.1. In the uppermost level, only target systems and users can be recognized. When an individual system is chosen to become accessible (for example *s1*), only the data flow diagrams (*dfd*) which specify that particular system become available. If the user chooses one of these diagrams to become accessible (say *dfd3*), only the functions, data flows, etc. of the chosen diagram will become available.

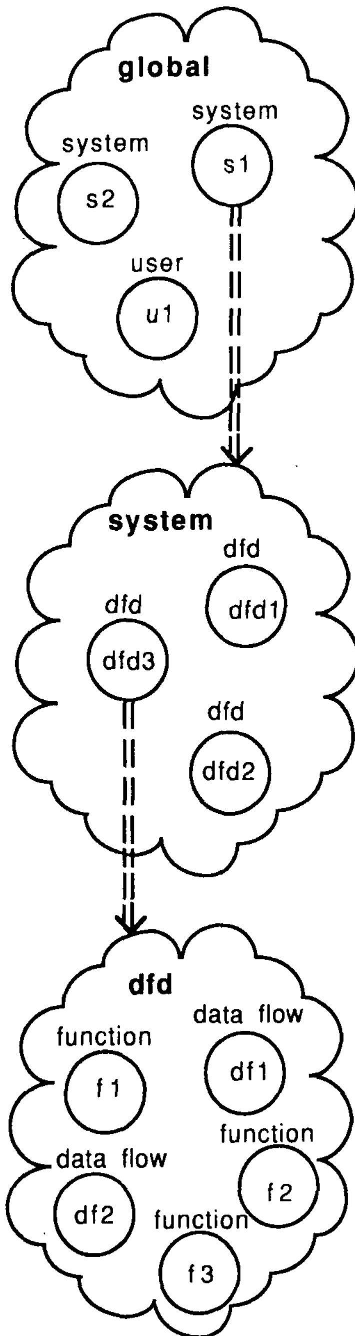

Figure 4.1: The Cluster Hierarchy for the Structured System Analysis

It should be noted that though any given aspect of a system may be refined, other aspects, which have not automatically been refined, continue to exist. In the above example, when the user chooses a particular system to become accessible (for example *s1*), information from other systems (*s2*) is not lost, but only detailed information of the chosen system becomes available. Only one cluster of objects of each cluster type is accessible at any one time.

4.4.2 Graphical Representation of ORM

Object types in ORM diagrams are represented by divided circles. The upper half of the circle is used to write the name of the object type, while the name of the host cluster associated with the object type is written in the lower half (see Figure 4.2).

Relationships always link two objects, termed the source object and target object of the relationship. A relationship is represented by an arc linking object types which are associated through the relationship type. In ORM, every relationship type has its correspondingly opposite relationship type. Thus, each relationship type is joined to its opposite relationship type by an oblique line; the name of the relationship is written close to the corresponding arc in the side, forming an acute angle with the oblique line. Every relationship type contains a maximum and minimum mapping, which characterize the quantity of relationships of a particular type in which the source object may be involved.

Figure 4.2 shows an example of ORM usage. It models two object types (*author* and *book*), which are related to each other by the relationship type *writes*, whose opposite relationship type is *written by*. These object types are within the *library* cluster type.

Author has the attribute types *address author* and *number published books*, and *book* has the attribute types *published date* and *publisher.* In this example, as far as mappings of relationships are concerned, there should be at least one *author* who has written a *book* (1-m), and at least one *book* in a *library* which has been written by a particular *author* (1-n).

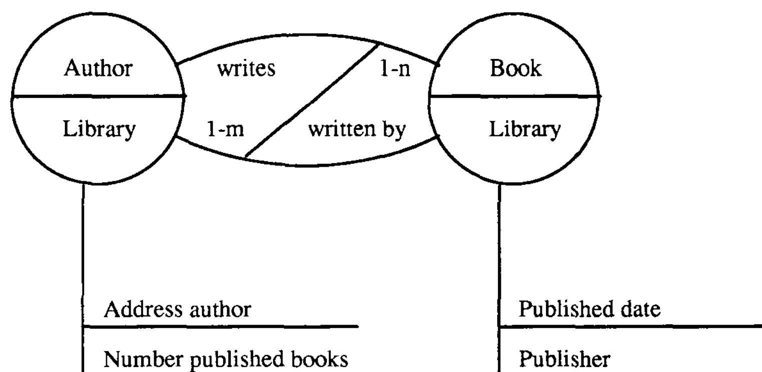

Figure 4.2: Example of use of ORM

4.4.3 Advantages of ORM

In this research work, ORM has been employed in order to analyse the requirements of the software maintenance process and to design a solution to this problem. It has been chosen as it provides some desirable and appropriate characteristics for the task. These characteristics yield benefits such as:

- The provision of a visual approach to information representation (diagram), which is able to convey a considerable amount of information.
- A mechanism which enables the grouping of the real-world entities into smaller clusters, facilitating the implementation of version control and the notion of scheduled releases.

- The implementation of the model can easily provide comprehensive analysis capabilities, which include traceability, consistency and completeness checks.

An example of formalization using ORM is given by Traina *et al* [110]. In their work, the methodologies which were automated and integrated into a software development environment named SIPS (Integrated System for Software Production) were formalized using ORM. Additionally, in the Capretz's work [25] the SCM discipline was formalized in order to be automated and integrated into SIPS. The formalization involved the creation of a generic model of SCM using ORM. Since SIPS is an environment for software production, it mainly provides support for the development phases of the life-cycle. Therefore, the SCM discipline applied to this environment aimed to provide support for the control of the generation of new SCIs and changes applied to these SCIs. However, the work did not deal with the particular problems of software maintenance, as has been done in this thesis.

4.5 Summary

In this chapter, some well-known data modelling techniques, along with their merits and deficiencies, were briefly surveyed so as to illustrate the work being carried out in this area and to provide a basis of comparison with the model used in this thesis. The conceptual model for the proposed method will be defined using ORM. The ORM formalism was described, with particular emphasis laid upon the features employed by this work.

Having compared ORM to other models described in this chapter, it can be con-

cluded that the former is an enhanced semantic model. ORM cannot be fully classified as an object-oriented model because, according to some authors [17, 68], the behaviour capabilities must be inherent in the model, which is not its case. ORM provides capabilities for dealing with the representation of data, ignoring the behavioural aspects. Nonetheless, as claimed by some authors, the latter characteristic is very much an implementation aspect of which ORM is capable of distinguishing. Therefore, ORM is independent of implementation aspects, and can be seen as a mechanism to aid the formalization of the method. In the object-oriented approach, the operations are closely associated with each object type; i.e., each object type is connected with its specific operations, which represent the activities to be performed on objects of that type. ORM, however, does not enforce this characteristic. Moreover, inheritance is another characteristic found in object-oriented models which ORM also does not take into account.

The next chapter details the proposed method; its formalization using ORM is described in Chapter 6.

Chapter 5

COMFORM - A New Method for Software Maintenance

The previous chapters presented some aspects of the maintenance of existing software systems. Various existing models of software maintenance have been discussed and their strengths and weaknesses have been pointed out. Despite increasing recognition that maintenance is a major problem during the life-cycle of software systems, there are still diverging opinions about how to tackle this problem. This chapter presents a new method for software maintenance called COMFORM (COnfiguration Management FORmalization for Maintenance).

5.1 The Objectives of COMFORM

COMFORM [23, 24] aims to provide guidelines and procedures for carrying out a variety of activities during the maintenance process by establishing a systematic approach to the support of existing software systems. The proposed method accommodates a change control framework around which the Software Configuration Management (SCM) discipline is applied. It aims to exert control over an existing software system whilst simultaneously incrementally redocumenting it.

The major activities carried out by COMFORM are:

- Acceptance of change proposals and analysis of their viability for implementation.
- Enforcement of evaluation of approved changes in terms of costs and resources.
- Promotion of scheduled system releases, so that the maintenance process can be planned and organized.
- Specification of the maintenance task by grouping together the modifications of a system release.
- Provision of managerial and technical reports.
- Enforcement of documentation of software components which require modification.
- Enforcement of the application of software quality by assuring completeness, consistency and traceability of existing software systems.
- Provision of audit trails, historic and current status of changes.

- Provision of guidelines for revalidating existing software systems.
- Provision of input to project management and quality assurance systems.

A change control framework has been established in COMFORM in order to preserve software quality. The Software Maintenance Model (SMM) institutes this framework, which aims to systematize the software maintenance process by specifying the chain of events and the order of stages that a change has to go through. The output of the SMM phases are represented by forms which allow a methodical approach towards the establishment and control of traceability throughout the maintenance process. These forms are the source of documentation of maintenance history and system redocumentation. The use of the SMM forms results in considerable advantages, such as:

- Providing a uniform structure of documents of software systems under COMFORM, since forms are pre-defined. Such a uniformity of information avoids inconsistency and unnecessary differences.
- Facilitating the application of SCM techniques, since it is easier to control pre-defined documents.
- Easing completeness checks, by ensuring that no essential details are omitted.
- Easing consistency checks, by ensuring that the information required by a form is provided by other forms in the configuration.
- Facilitating traceability between phases by establishing the relationships between components of different phases in the forms.

The SCM discipline is central to the development of COMFORM since it is concerned with the development of a set of procedures and standards for managing evolving software systems. In essence, it is concerned with change: how to control change, how to manage software systems which have been subject to change, and how to release these changed software systems to the users. Moreover, the objective of COMFORM is not just to develop a method to change programs in order to fix an error or to implement a modification, but also to improve the future maintainability of the software systems being maintained. Without the application of such a discipline, it is easy to send out a wrong or bad version of a software component. A primary goal of applying this discipline to the method is to improve the ease with which changes can be accommodated, thereby reducing the amount of effort expended on maintenance.

This chapter concentrates on the method underlying COMFORM. In consequence, the SMM framework is presented, followed by the forms which represent the outcome of the phases. The role of the SCM discipline, and each of the SCM functions, as they relate to COMFORM, are also discussed.

5.2 The Software Maintenance Model

The Software Maintenance Model (SMM) aims at improving the flow of maintenance activities by providing guidance throughout the maintenance process, and determining the organization and content of the information needed to support these activities in COMFORM. The SMM differs from other models for the software maintenance process, as already discussed in Chapter 3, because it is approached from the SCM

viewpoint.

The SMM identifies the activities undertaken during software maintenance and the information needed or produced by these activities. The model is based on the traditional waterfall life-cycle model of software development. This is a convenient approach, because it allows the process to be represented in a graphical and logical form, providing a framework around which quality assurance activities can be built in a purposeful and disciplined manner.

Being a maintenance model, the SMM highlights the considerable influence of the existing software system on this whole process. The outcome of each SMM phase is a completed form which represents a point in the maintenance process. These completed forms are, therefore, the natural milestones, i.e. the baselines of the software maintenance process, and offer objective visualization of the evolution of that process.

The following phases comprise the SMM:

1. Change Request
2. Change Evaluation
3. Maintenance Design Specification
4. Maintenance Design Redocumentation
5. Maintenance Implementation
6. System Release

Figure 5.1 represents the SMM. In this Figure, the rectangles represent the SMM

phases and the ovals represent the baselines formed from the output of the phases. Being a software maintenance model, it is essential that the influence of the existing software system on the process should be considered. For this reason, the *change evaluation* phase has been introduced, during which modifications are considered in relation to the existing software system. The need to understand the existing software system is motivated by the information required by the incremental redocumentation process. Understanding will be facilitated by the information contained in the different forms as changes take place.

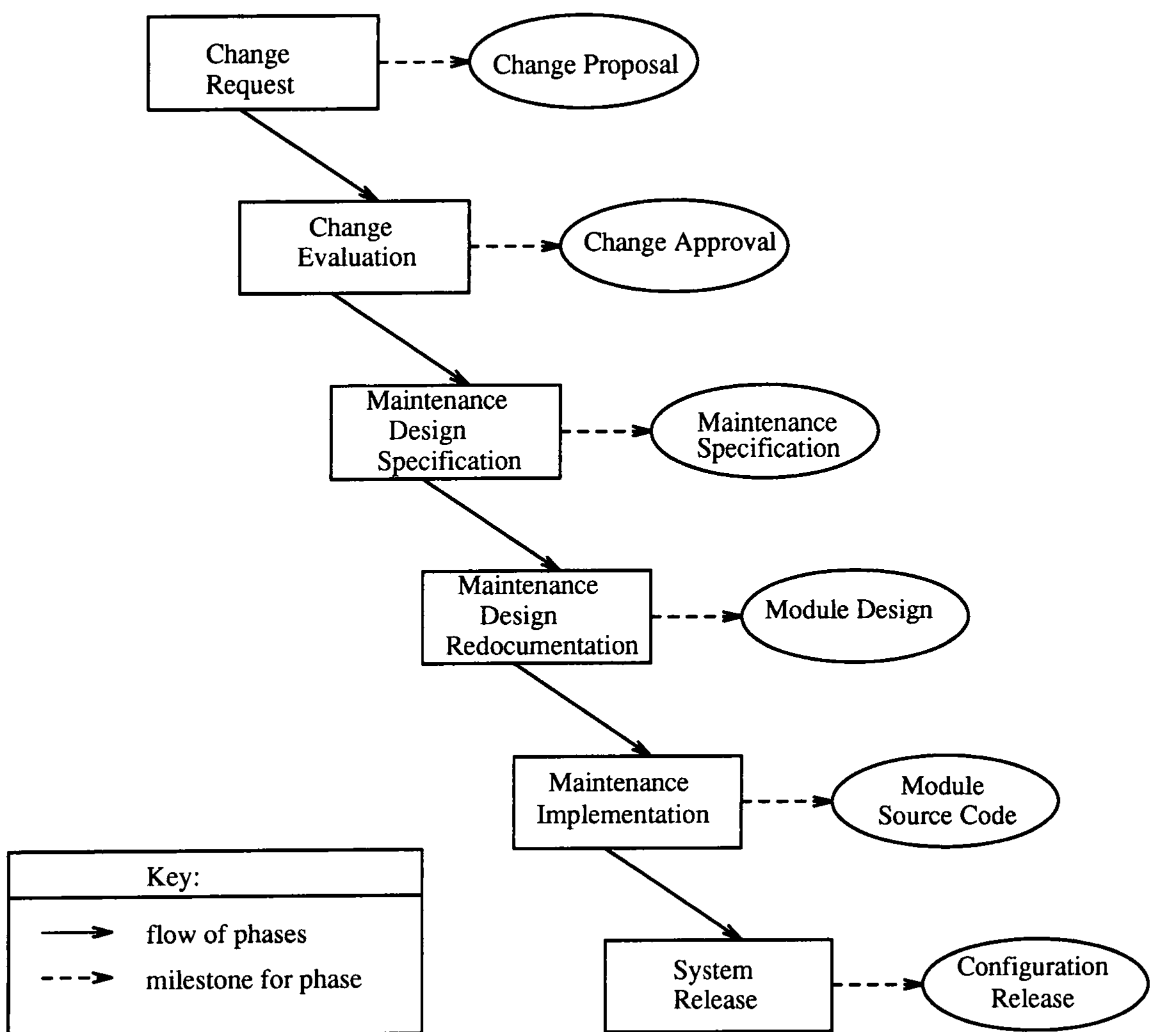

Figure 5.1: The Software Maintenance Model

Although there is no explicit re-verification phase in the model, all the activities

associated with this phase are incorporated in the method for establishing the baselines. For example, the Configuration Release form has the fields *Integration tests outcome* and *System tests outcome*, which have to be completed before the *Configuration Release* baseline can be established.

As with models of software development, SMM phases may overlap. Also, it may be necessary to repeat one or more phases before a modification is completed. However, the products which represent the output of the phases must constitute a baseline, and cycling must be controlled. The output of SMM phases is a number of predefined forms which consist of three sections (*identification*, *status* and *information*), as depicted in Figure 5.2.

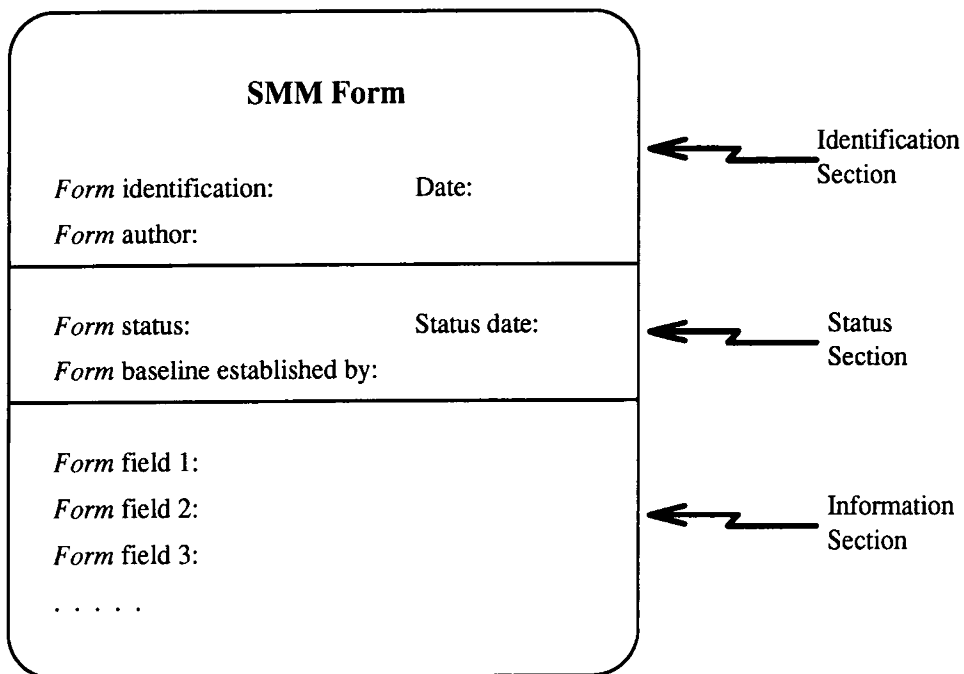

Figure 5.2: The Pattern of SMM Forms

The *identification* section contains the basic information about a particular SMM form: the form identification (to allow every single form to be traced throughout the maintenance process), the date of its creation, and the author who created that particular form.

The baselines of SMM forms are established by the satisfactory outcome of a quality assurance process on the completed forms (described in greater detail in Subsection 5.3.4.2) for each of the phases. The control of baseline establishment is common to all forms and documented in the *status* section of each form. The first field of this section is an indicator of the current status of the form. The possible values for this field are: *in development*, *effective* and *frozen*. The initial status of the forms is *in development*. The status changes to *effective* after its initial evaluation (described in greater detail in Subsection 5.3.4.1). The final status of a form is *frozen*, when its baseline is established. The second field records the date on which the current status has been assigned to the form. When the status of the form is *frozen*, the last field records the author who established the baseline.

The *information* section is specific to each SMM form and contains data concerned with particular phases of the model. The fields of this section will be detailed subsequently, as the following subsections give further details of each particular phase.

5.2.1 Change Request

All requests for software maintenance will be presented in a standardized manner. The Change Proposal (CP) form is the form associated with this SMM phase. The filling-in of a Change Proposal form triggers the process of maintenance in COMFORM. The form contains the basic information necessary for the evaluation of the proposed change. If the proposed change is for corrective maintenance, then a complete description of the circumstances leading to the error must be included. For the other types of maintenance, an abbreviated requirements specification must be

submitted. Figure 5.3 shows the basic information for any type of change proposal.

Change Proposal

CP identification: Date:
CP proposed by:

CP status: Status date:
CP baseline established by:
(Reason for abandoning:)

1. CP description:
2. Reason for change:

Figure 5.3: The Change Proposal form

While the information contained in the *identification* section of this form is as described above, the *status* section for this particular form incorporates one additional field: *Reason for abandoning.* The *CP status* field is an indicator of the status of the proposed change in the maintenance process. The initial status is *in development* when the change is assigned for evaluation. That status changes to *effective* after its initial evaluation. The final status is *frozen* when the change proposal may have been approved and a corresponding Change Approval form has been generated in order to carry on the change evaluation. The proposed change may also be rejected (which will also be *frozen*), and is then followed by filling in the field *Reason for abandoning.* The field *Status date* shows the date the current status was assigned to the form. Where the status of the form is *frozen*, the author is associated with the *CP baseline established by* field.

In the *information* section, the field *CP description* provides a short functional or

technical description of the proposed change so that it can be evaluated and then approved or rejected. The field *Reason for change* is a brief description of the benefits of carrying out the change. This is used to help the maintenance staff in ranking and approving the proposed change. The justification for corrective changes is straightforward. Usually, the change will be made on the justification that the program must function correctly. However, in some cases, the cost of the change or its effect on the rest of the software system may be so enormous compared to the minor inconvenience resulting from the failure, that the user may choose to tolerate the failure rather than risk introducing new problems or modifying present operating procedures. The justification for adaptive, perfective and preventive changes may be more complicated, since the benefits may be difficult to evaluate compared to the cost of implementing and the risk of degrading quality.

5.2.2 Change Evaluation

In the *change evaluation* phase, the maintainer is primarily concerned with understanding the change, and its effect within the software system. An accurate change diagnosis is performed to assess the feasibility of the proposed change in terms of cost, resources, and schedule, resulting in approval or rejection. The rejected proposed change is then abandoned. If the proposed change is approved, a corresponding Change Approval (CA) form is created and its evaluation continued. The inadequacies, or unfulfilled requirements described in the Change Proposal form are identified in the existing software system. In addition, every software component involved in the proposed change must be known. If the information contained in the Change Approval form is comprehensive then the maintainers are able to react

quickly to problems, analyse enhancements properly, evaluate impacts and estimate resources. In this phase, the approved changes are ranked and selected for the next system release. The changes are batched by system releases and the work is scheduled. The result of this phase is the Change Approval form depicted in Figure 5.4.

Change Approval

CA identification: Date:

CA authorized by:

CA status: Status date:

CA baseline established by:

1. Related CP:
2. Type of change:
3. Identification of change:
4. Involved sw components id.:
5. Resource estimates for change (design):
6. Resource estimates for change (coding):
7. Resource estimates for change (testing):
8. Priority of implementation:
9. Consequences if not implemented:

Figure 5.4: The Change Approval form

The Change Approval form is one of the documents used as the basis for planning the system release. It is a vehicle for recording information about a system defect, a requested enhancement and quality improvements. The Change Approval form, along with its corresponding Change Proposal form, is the basic tool of a change management system. By documenting new software requirements or requirements that are not being met, these forms become the contract between the person re-

questing the change and the maintainers who work on the change. The field *Related CP* ties the Change Proposal form to its corresponding Change Approval form.

The field *Type of change* classifies the work as perfective, adaptive, corrective, or preventive maintenance. The information contained in the field *Identification of change* shows the maintainer something about the nature of the work, and indicates to management how maintenance time is going to be spent. This field provides a more elaborate functional and technical description of the approved change, which is dependent on the type of maintenance as outlined below:

- *Perfective Maintenance* - Identify new or altered requirements.
- *Adaptive Maintenance* - Identify the change in the environment.
- *Corrective Maintenance* - Identify repeatable error symptoms.
- *Preventive Maintenance* - Identify the deficiency in performance, maintainability, etc.

The field *Involved sw components id.* provides information about which software components are directly involved in a change. The ripple effect of a proposed change is necessary to estimate accurately the scope of work and the resources required. Once the impact and ripple effect of a change are determined, preliminary resources estimates can be developed. These estimates are approximations of the work required to accomplish the changes in all the involved parts of the software system. Estimates can be expressed in any unit of measurement meaningful to the organization, such as hours, days or weeks. The fields *Resource estimates for change*, which apply to design, coding and testing, record these resource estimates for a proposed

change. These estimates are important to project management and can also be used as references for similar work on other projects, or future system releases.

The field *Priority of Implementation* indicates the time frame necessary for completion of the task. This classification is important as it highlights the magnitude, criticality, or complexity of change proposals. The priorities are classified as:

1. *Critical* - Where repairs should receive immediate attention ahead of any currently scheduled system release.

2. *Important* - When the software system is operational and can be manually overriden or ignored until a specific date. The proposed changes may upgrade to *critical* priority if the problem is not fixed by the date required.

3. *Minor changes* - When the proposed repairs and enhancements can be deferred until the next system release, but time and resources permitting, should be performed for this release.

4. *Optional* - Includes slight repairs or enhancements that may be worked out in the next system release, as resources and time allow.

Critical approved changes should be attended to immediately by the maintenance staff. Other changes may be collected for periodic review by a Change Control Board. This board includes user representatives as well as members of the maintenance organization. The responsibilities of the board include deciding the fate of change proposals based on long-term goals of the organization, user needs, cost considerations and the batching of approved changes by system releases. Before being reviewed by the board, change proposals are studied by the maintenance staff to determine the resources (effort and time) needed to make each change, the impact

of the change on other software components, and the cost of the change. The field *Consequences if not implemented* is then filled in by the maintenance staff during this preview analysis, in order to provide more information to the board.

5.2.3 Maintenance Design Specification

This phase is characterized by the structure of the modification, which is in the form of a complete, consistent and comprehensible common specification of all the changes which should be made. In addition, *how* the software components have to be modified is clarified. All the related approved changes selected for the next system release will be in the same maintenance specification. The design of a modification requires an examination of the side effects of changes. The maintainer must consider the software components affected, and ensure that component properties are kept consistent. The integration and system tests (to be performed during the revalidation process in the *system release* phase) need to be planned or updated. In addition, if the change requires a new logic or new features to be added to the system, then these have to be specified and incorporated. In this phase, the activities to be performed are detailed, in order to work out the alterations necessary to implement all the related approved changes (which have been batched for the next system release).

The boundary between the *change evaluation* and the *maintenance design specification* phases may at times seems blurred, because the analysis and review performed in the *change evaluation* phase sometimes overlaps with the specification of the change. Nevertheless, this phase is more concerned with generating a common specification for all the changes proposed and approved for a planned schedule release.

The resultant form for the *maintenance design specification* phase is the Maintenance Specification (MS) form, represented in Figure 5.5.

Maintenance Specification

MS identification: Date:

MS formulated by:

MS status: Status date:

MS baseline established by:

1. Related CAs:
2. Specification of change:
3. Affected sw components id.:
4. Consequences of change:
5. Other necessary changes:
6. Integration tests:
7. System tests:

Figure 5.5: The Maintenance Specification form

The Maintenance Specification form is generated after the approved changes for the next system release have been selected. The relationship between these selected changes and their corresponding specifications are detailed in the field *Related CAs* of the created Maintenance Specification form.

In the specification of the proposed change, the different types of maintenance require different ways of specifying the change. The field *Specification of change* should take these differences into account as described below:

- *Perfective Maintenance* - Specify operation of the enhanced system.

- *Adaptive Maintenance* - Revise the specification to reflect the change and adapt the original specification to the new system environment.
- *Corrective Maintenance* - Specify correct operation of the software system.
- *Preventive Maintenance* - Specify the desired performance or quality standard and/or redesign a distinct subsystem to satisfy the same specification, so that it uses less resources, or is better structured and maintainable.

The field *Affected sw components id.* provides information about which software components are indirectly affected by a proposed change. Such components often come up during the impact analysis of the proposed changes. In the case where several software components are affected by the change, resources estimates may have to be reviewed and the schedule for the next system release be updated.

The field *Consequences of change* reports the side effects encountered during the impact analysis of the proposed changes. The optional field *Other necessary changes* may have to be filled in if, during the specification of the approved changes, unpredicted changes have to be performed in the software system in order to fulfil the original proposed changes. For instance, if during the impact analysis, the proposed change affects software components not predicted in the analysis and review, new change proposals have to be elaborated and introduced to the formal change control procedures established by COMFORM. In such instances, this field is filled in by establishing the relationship between the current maintenance specification with new Change Proposal forms.

Tests plans should be elaborated at this stage. They should be based on impact analysis and describe how and when the software system should be tested. The

fields *Integration tests* and *System tests* should contain the tests to be carried out in the software system in order to complete the next system release. For instance, during the integration tests (with unit test completed) only the interfaces between the modified software components need to be examined, whereas during the system tests, only the interfaces between users and the modified software components need to be examined.

5.2.4 Maintenance Design Redocumentation

This phase, along with the next SMM phase, facilitates system comprehension by incremental redocumentation, as proposed by the method. The forms associated with these two phases aim at documenting the software components of an existing software system under COMFORM. Therefore, the forms will be filled in when the corresponding software component has to be modified. The software components that should be changed are re-defined and the new components which might appear must be specified.

The algorithms and the behaviour of procedures for both normal and exceptional cases are also explained by this phase. In addition, the tests for each of the changed or implemented software components are planned. The form associated with this phase is named Module Design (MD) form, as depicted in Figure 5.6. The information contained in this form is independent of the maintenance category being performed.

COMFORM proposes a one-to-one relationship between a Module Design form and an existing software component. The purpose of such a relationship is to capture a

Module Design

MD identification: Date:

MD designed by:

MD status: Status date:

MD baseline established by:

1. Module purpose:
2. Algorithms outline:
3. Interface definitions:
4. Test plans:

Figure 5.6: The Module Design form

higher level documentation of these components, while maintaining them. The aim of the Module Design form is to provide some documentation to enhance readability and to convey more clearly the software component meaning. Thus, the field *Module purpose* contains an outline of the purpose of the software component associated with that Module Design form. In the *Algorithms outline* field, the limitations, restrictions and algorithmic idiosyncrasies of a software component are recorded. The current input and output interfaces of a software component are kept in the field *Interface definitions.* Unit test plans should be elaborated in this phase and kept in the field *Test plans.* During unit test, only the revised software component and the specific changes need to be examined.

5.2.5 Maintenance Implementation

The purpose of COMFORM is mainly concerned with building up maintenance history and devising an abstraction of the operational system. Therefore, the proposed method does not deal directly with the source code itself. In this phase, SMM forms have a link with the components of an existing software system. This is the phase in which the system source code should be actually changed and new components implemented. The coding standards and other conventions specified for this phase should be followed. The implementation should exploit programming language features (such as structuring facilities, user-defined types and assertion statements) to state properties and dependencies, and encourage modularity and encapsulation.

The SMM form associated with this phase is the Module Source Code (SC) form, and like the Module Design form, its information is independent of the maintenance category being performed. A Module Source Code form is depicted in Figure 5.7.

There is one Module Source Code form related to each software component and a one-to-one relationship between each Module Source Code and Module Design form. The information contained in each of these forms complements the other. The first field of the Module Source Code form (*Corresponding MD*) formally links the forms of these two phases. While the Module Design form is aimed at keeping general and stable information for a software component, its corresponding Module Source Code form aims at keeping the information pertaining to modifications performed in these components. Hence, the field *Tests outcome* records the result of the unit test performed after a software component has been modified. Further, the field *Comments* may contain any remark detected during the implementation of the modification which is worth documenting. The last field, *SC understood by*, records the name of

the maintainer who has some knowledge about that particular software component.

Module Source Code

SC identification: Date:

SC implemented by:

SC status: Status date:

SC baseline established by:

1. Corresponding MD:
2. Tests outcome:
3. Comments:
4. SC understood by:

Figure 5.7: The Module Source Code form

5.2.6 System Release

System release is the last phase of the SMM before a new configuration containing the approved changes is released to the users. Validation of the overall system is achieved by performing the integration and system testings on the system.

Once modifications on the system have been performed under the configuration control function, the task at this stage is to certify that all baselines have been established. The release of the new configuration should be followed by informing the interested and/or related members of the group (users, managers and maintainers) about the requested changes completed in the configuration. The Configuration Release (CR) form, represented in Figure 5.8, contains details of the new configuration.

Configuration Release

CR identification: Date:

CR created by:

CR status: Status date:

CR baseline established by:

Comprises:

CPs:

CAs:

MSs:

Is composed of:

1. Integration tests outcome:
2. System tests outcome:
3. Configuration distributed to:

Figure 5.8: The Configuration Release form

A Configuration Release form is the system release planning document, which aims to keep the information pertaining to the history of a maintenance phase. The fields *Comprises CPs*, *CAs* and *MSs* are concerned with retaining the connection with the Change Proposal, Change Approval and Maintenance Specification forms which constitute that particular system release. A Configuration Release form also shows the Module Design and Module Source Code forms that have been modified during that particular system release. Such links are displayed in the field *Is composed of* at the end of the development of the corresponding Configuration Release form. Additionally, in order to release the system to users, the integration and system tests must be performed. The outcome of such tests must be documented in the fields *Integration tests outcome* and *System tests outcome* respectively. The successful outcome of these tests enables the Configuration Release form to have its baseline established. Those users who are to receive the current system release are registered in the field *Configuration distributed to.*

5.3 The SCM Discipline Applied to COMFORM

In the previous section, SMM phases have been described. The emphasis was on the description of each type of form and their fields, together with their respective meanings and functions in the proposed method.

The specific functions which the SCM discipline plays in the whole process are detailed in this section. The SCM discipline provides the formal mechanism to establish the baselines of the method. Such baselines can only be changed through the formal change control procedure. In the following subsections, details of the

application and guidance provided by the four functions of the SCM discipline on COMFORM are described.

5.3.1 Software Configuration Identification

The purpose of this function is to highlight the constituent parts of a software system in a manner that makes explicit the relationship between these parts. It is the process by which the pieces of an existing system are transformed into a structured entity, thus identifying the Software Configuration Items (SCIs) [13]. Therefore, this is the first function which should be carried out in COMFORM so that the SCM discipline can be applied to the whole maintenance process determined by the proposed method. The SCIs in COMFORM have already been identified and defined as the SMM forms. The effect of this function is to provide the structure of links and dependencies between the forms. This also helps during the task of obtaining the components involved in, and affected by, a required change. As a result, the SCM discipline is able to control the release and changes in the forms throughout their existence, record and report their status, and verify their completeness and correctness.

5.3.2 Software Configuration Control

The software configuration control function in COMFORM is concerned with the control of changes made in an existing software system, from the change proposal, evaluation, approval and implementation, to its release. It is also concerned with

the establishment of standards for software maintenance. Therefore, in the context of this work, its role is to ensure that any change required in a software system is defined and implemented by following SMM phases, which institute a change control procedure to monitor changes. In so doing, change information is gradually recorded by the filling in of forms. This procedure ensures that all work performed to implement a change is traceable to change proposal and that changes to a software system can only be made by properly authorized maintainers. It also prevents unauthorized changes from being made, since no implementation can proceed without authorization. Since the changes are traceable to the original change proposal, the auditing process can check that only the approved requested changes have been made in the software components. This SCM function also controls the versions of SMM forms. The version control in COMFORM is further detailed in Section 5.4.

5.3.3 Software Configuration Status Accounting

The configuration status accounting function aims at recording and reporting the current status, as well as the evolution of the existing software systems. The information necessary to perform this function is incrementally obtained as the SMM forms are filled in as a result of changes to the software system. The implementation and effective use of this function in COMFORM is achieved by its automation and by the supporting information contained in the SMM forms.

A number of reports can be obtained so that answers can be provided for a variety of queries about the existing software systems kept under COMFORM. Such reports can be of help either to project managers or maintainers, by showing information such as productivity, or providing the history and current status of software systems.

Typical queries about the system's managerial aspects might be:

1. Which software systems are under COMFORM?
2. How many system releases of a particular software system have been created?
3. What is the number of change proposals made per software component?
4. What is the number of change proposals made per maintenance category?
5. How many and which software components are being modified per maintainer?
6. What are the total resources spent on incorporating a particular change proposal in a system release?
7. What are the total resources spent on designing, coding and testing each maintenance category of a particular software system?
8. Which Module Design forms have been documented by a particular maintainer?
9. For each Module Source Code form, which maintainers understood the software component?
10. What are the change proposals requested by a particular user?
11. Which users have taken delivery of a particular system release of the software system?
12. How many change proposals are outstanding on a particular software system?

Typical queries involving technical aspects which can provide help to maintainers are:

1. How many change proposals are grouped in a particular system release?
2. What are the creation and release dates, and the status of a system release?
3. Which change proposals are related to a particular software component?
4. Which Module Design forms are incomplete (not *frozen*) in a particular system release?
5. How many times has a software component been changed (number of versions) and what are the reasons for these changes?
6. Which forms are related to a particular system release?
7. What is the current status of the system configuration to be released?
8. Which change proposals of a particular software system have been rejected?
9. Which maintenance specifications are related to a particular software component?
10. What is the system configuration released on a particular date?
11. What are the versions of a particular software component?

5.3.4 Software Configuration Auditing

The traditional SCM auditing function comprises the processes of verification and validation [21]. The verification process is largely administrative, basically consisting of performing checks on the outcome of phases, in terms of correspondence and traceability to the previous baseline. It also ensures that the correct and current versions of all product parts are included in the baseline. The validation process

involves a technical assessment of the baseline, i.e., validating it against the change proposal requirements. It may also involve regression tests to help confirm the absence of unanticipated side effects in functions not related to those being modified.

The auditing function within COMFORM is the process which determines the overall acceptability of the proposed baseline at the end of each SMM phase. This process aims to establish the baselines of SMM forms. It uses the related product assurance disciplines of test and evaluation, completeness, consistency and quality assurance.

The auditing process in COMFORM can be divided into two separate stages. The first stage basically consists of performing checks on SMM forms for the completeness and correctness of the information. The checks are specific to each SMM form, and consequently differ from one SMM form to another. The outcome of the software tests which have been performed are also checked at this stage. The specific checks to be performed in each SMM phase are detailed in Subsection 5.3.4.1. The second stage aims at establishing the baselines of all the forms involved in a system release; thus, consistency and traceability between SMM forms are checked. Further details of this stage are given in Subsection 5.3.4.2.

5.3.4.1 Initial Evaluation

The basic checks performed during the first stage are dependent on each SMM form. The result of this stage alters the status of the forms from *in development* to the intermediate status *effective*. In order to upgrade the Change Proposal form, the proposed change should be evaluated and the decision, whether or not it should be implemented or abandoned, must be taken.

The checks applied to the Change Approval form require that the proposed change must be clearly understood and identified by specifying the software components involved in the modification, as well as stating the resources estimates, schedule and priority of implementation.

In the *maintenance design specification* phase, a Maintenance Specification form should provide a complete, unambiguous and comprehensible specification of all the changes of a system release. In addition, the integration and system tests must be elaborated and the other necessary changes (if any) should also be included in the Maintenance Specification form.

At this stage, the Module Design forms must be filled in, thus assuring that the documentation of the software components, which have undergone modifications, is improved. Additionally, the individual tests of those software components should be specified. This phase may also involve the simulation of the software design, in order to validate the software system with the end users or to check for requirements which are not covered by the current change design.

During the *maintenance implementation* phase, it must be assured that individual tests of the modified and/or newly implemented software components are performed to check their functionality and interface standards. Futhermore, checks that the coding standards are in compliance with the standards defined by the software configuration control function must be made.

In the *system release* phase, checks should ensure that the integration tests which exercise the modified functions or subsystems have been performed. These tests should be carried out independently and in controlled combinations. Checks should also make certain that system tests have been performed, to verify that the software

system meets its specified requirements. Additionally, the use of correct versions included in the system release to be completed must likewise be verified.

5.3.4.2 The Establishment of Baselines in COMFORM

The establishment of baselines is the most important step to be carried out at the end of each SMM phase. Baselines are established from the successful outcome of the application of quality assurance procedures to the SMM forms. The result of this stage alters the status of the forms from *effective* to the permanent status *frozen.*

Establishment of baselines is performed after the initial evaluation of the auditing process, and represents the moment when all the consistency and traceability checks have been performed. The essence of this stage is to perform checks, not on the individual forms, but on the set of SMM forms, which represents a system release. Therefore, this process has a sequence which should be followed, in order to make certain that the specified forms are in accordance with each other, and to assure the integrity of data being manipulated during a system release.

The first stage of the process of establishing the baselines is represented by the Module Source Code form, since it is the nearest form which links the software component to COMFORM. Therefore, to have their baselines established, the forms of this category must be documented and be in accordance with the corresponding software component. The second stage of establishing the baselines involves the Module Design forms, which represent a higher level of documentation of those software components.

The documentation of the maintenance process is obtained from the Configuration

Release, Change Proposal, Change Approval, and Maintenance Specification forms. As a result, the sequence by which the baselines are established should follow the sequence of SMM phases. Hence, the Change Proposal form is the first one to have its baseline established, which means either the Change Proposal form has been approved and the maintenance process should be carried on, or the proposed change should be abandoned after having been rejected. The consistency check enforces the one-to-one relationship between an approved Change Proposal form and its corresponding Change Approval form.

The Change Approval form and the Maintenance Specification form should be filled in independently of each other, but obviously the Change Approval form has its baseline established first. Such events only happen if the corresponding forms of the involved and affected software components have already had their baselines established. After this point, it is presumed that the implementation has already been carried out, so the next step involves the final evaluation of the system release. In this case, the Configuration Release form should be filled in with the results of the testing and have its baseline established.

At this point, the new system release is ready to be made available to the users, with the assurance that all the software components required for maintenance activity have been documented, and that the maintenance applied to the software system has been monitored and controlled.

5.4 The Version Control in COMFORM

Software systems usually have many different versions and consist of several software components, which in turn may have many different versions. It is useful to keep track of different versions of these systems, as well as the versions of the software components which make up a particular release of a software system. COMFORM aims to supply this capability by providing a mechanism to manipulate versions of SMM forms, and to enforce restrictions on the evolution of such forms, so that this evolution process is observable and controllable.

Babich [5] distinguishes between two types of versions: *revisions* and *variations.* COMFORM adopts both these concepts, but a version (revision or variation) of a form is only created after undergoing the quality assurance process (defined to establish the baseline) of the corresponding SMM phase. Hence, a version of a form is under SCM control, and it can no longer be changed. If other changes to this version are required, a new version should be created.

A *revision* of a form is a new version, created to supplement previous ones. A form may have many revisions, reflecting its evolution. Each successive revision should denote the removal of existing software errors (corrective maintenance) or in some other way, the improvement of earlier revisions by either adding functionality (perfective maintenance) or improving the quality of the software being maintained (preventive maintenance). Consequently, revisions are in a linear order, being related to the time sequence in which they are created.

A *variation* of a form fulfils a similar function for slightly different situations and therefore acts as distinct versions for the same form. Multiple variations of a form

may coexist as equal alternatives. Unlike revisions, there is no meaningful linear order among variations. The intent of creating variations is to support coexistent alternatives, such as different types of hardware (adaptive maintenance) and alternative functionalities (adaptive and perfective maintenance).

All versions of a form are related to each other by being either revisions or variations. There may be both variations and revisions of a single form, as a variation may require software error corrections and performance improvements, resulting in multiple revisions of a variation.

Since versions represent the satisfactory outcome of the quality assurance process of SMM forms, other ways of controlling their evolution before they are under SCM control need to be adopted. For this reason, as in [60], SMM forms can be in one of the three status groups: *in development*, *effective* or *frozen*. A version of a form (revision or variation) is obtained when its status is *frozen*. Therefore, before reaching the *frozen* status and being under SCM control, they are called *alternatives*. Alternatives have to be developed and audited in order to reach the *frozen* status. When alternatives are being developed, their status is *in development*. At this stage, maintainers can make as many modifications as required to the fields of the alternatives. Once the development or modification of an alternative has been finished, its status becomes *effective*. At this point, modifications to this alternative cannot be made and the quality assurance procedures are applied in order to perform the SCM auditing function. The successful outcome of the application of such procedures generates a version of a form; whereas, its failure returns the alternative to *in development* status, in order to allow the necessary changes to be performed.

The SMM forms which are first created during a system release are called *original*

SMM forms. These are created as a result of the formal procedures established by COMFORM, having their baselines established at the end of a system release. Once baselines have been established, the forms representing them cannot be changed. If modifications are required, alternatives should be created and controlled. Such alternatives are created according to the *versioned* SMM form, which is depicted in Figure 5.9. A versioned form (like an original form) also contains three sections termed *identification*, *status*, and *information*.

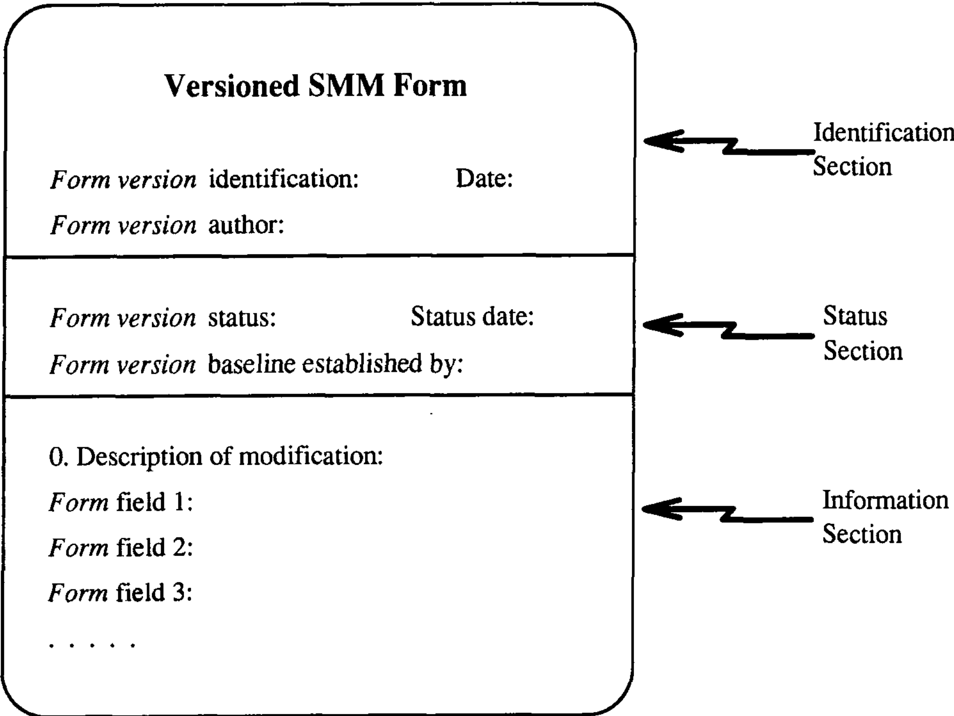

Figure 5.9: The Versioned SMM Form

The *identification* and *status* sections of a versioned form are very similar to the original form, except for the fact that they contain information about an alternative/version.

Every versioned form for all types of SMM forms has an additional field in the *information* section. This field is called *Description of modification*, and its content documents the purpose of generating an alternative for a particular *frozen* form.

The remaining fields of the *information* section for every versioned form are the same as those for the corresponding original SMM forms.

The necessity to create versions of forms is demanded by modifications required to be applied to them after they have been *frozen*. In COMFORM, SMM forms do not receive the same treatment, as far as version is concerned, since their purpose in the method is different. The Change Proposal, Change Approval, Maintenance Specification and Configuration Release forms are aimed at documenting maintenance history, whereas the Module Design and Module Source Code forms are aimed at providing an abstraction of the operational product. Thus, the concepts of *revision* and *variation* are more suitably applied to the latter forms.

The Change Proposal form may have versions if it was once *abandoned*. In this case, the generation of a version allows abandoned change proposals to be put back into the software system. A Change Proposal form may also have a version in the case of discrepancies being found during its evaluation; consequently, corrections are required to be undertaken by the person who proposed the change.

Configuration Release forms may also have versions. A version of a Configuration Release form aims at linking configuration releases together, which are either dependent on each other or part of the same context. This concept allows, for instance, the creation of a version of a Configuration Release form to correct errors in the previous releases. In doing so, the set of versions of Configuration Release forms provides the history of system release evolution.

Versions of Change Approval and Maintenance Specification forms should only be created to correct inconsistencies generated during their implementation. Therefore, versions of Change Approval and Maintenance Specification forms will only

exist within a system release. Once a Configuration Release baseline has been established, versions of its Change Proposal, Change Approval and Maintenance Specification forms can no longer be generated. New modifications to them mean the creation of a new Change Proposal form, with its corresponding Change Approval and Maintenance Specification forms generating a new Configuration Release form.

On the other hand, as Module Design and Module Source Code forms are the means of documenting the software components, they may have various revisions and variations to reflect the system evolution. The semantics of the required change determines whether a revision or a variation of Module Design and Module Source Code forms should be created. It is during the *change evaluation* phase that such a decision has to be made. In the case where a proposed change requires the replacement of Module Design and Module Source Code forms, a *revision* of them should be created. On the other hand, if the proposed change is intended to act as an alternative to the existing Module Design and Module Source Code forms, a *variation* of them should be created.

5.5 Final Remarks on COMFORM

An informal model for software maintenance has been presented, initially without reference to versions. The relationship of the model to the SCM discipline was discussed and then the issue of versions was introduced.

In broad terms, the application of COMFORM starts when a change is proposed to an existing software system, and finishes with a new system release to the users.

Between these two stages there are phases which guide the maintainer throughout the software maintenance process. These phases are supplied by the SMM, with the addition of the corresponding forms for each of its phases. The SMM has been developed to improve the flow of maintenance activities.

The SCM discipline applied to COMFORM provides the mechanism for maintaining and controlling the various baselines throughout the development of the forms which constitute a system release. In essence, the SCM discipline embraces the whole change process. Hence, the application of this discipline contributes directly to software quality by identifying and controlling change, assuring the change is being properly implemented, and reporting change to others who may have an interest.

The proposed method can be used manually, but is very amenable to automation. A prototype environment which provides automated support for COMFORM is described in Chapter 7.

The next chapter deals with the formalization of COMFORM, using the ORM concepts detailed in Chapter 4.

Chapter 6

Formalization of COMFORM

This chapter demonstrates how the concepts of COMFORM (described in the previous chapter) are modelled using ORM. The model presented in this chapter is relevant to COMFORM, since it formalizes the method underlying COMFORM. Such a formalization facilitates the visualization of the method, and supports the automation of the activities performed during the method's process. The formalization also allows COMFORM to provide a uniform set of services to create, retrieve and manipulate instances of forms in a persistent object base. Moreover, the cluster mechanism of ORM enables the implementation of the scheduled release concept and the version control discussed in the previous chapter. It also allows the storage of information belonging to several existing software systems in the same object base. Furthermore, the implementation of the model can easily provide comprehensive analysis capabilities, which include completeness, consistency and traceability.

6.1 The Modelling of COMFORM using ORM

The modelling of the COMFORM method was an evolutionary process which entailed the inclusion of different types of objects in the model. Some of these object types were subsequently removed from the model. SMM forms have been chosen as the basic object types of the model because they represent the main and essential entities within COMFORM. In addition, SMM forms are the software configuration items that have to be under the control of the SCM discipline. Other entities in the method have also been tried out in the model. For example, people involved in the maintenance process (maintainers, users and managers) were included as object types in an earlier model, but were rejected because they made the model too cumbersome and made it difficult to visualize the main points of the method's formalization.

The fields of SMM forms are the attributes and relationships of the objects in the model. Like most of the fields of SMM forms, people involved in the maintenance process are attribute types associated with the forms and are not object types in the model. The attribute types of the model provide additional features associated with the forms. Each of the relationship types chosen for the model represents a check of consistency or traceability that can be obtained within the model. The attribute, together with the relationship types are the contents of the forms which have to be specified, in order to enable them to go through the completeness checks.

The definition of cluster hierarchy, discussed in Section 6.2, has also gone through a series of evolutionary changes to find a more convenient structure which fits the method, so that COMFORM could benefit from the cluster mechanism in ORM.

The modelling of COMFORM is presented in two stages, consisting of the main model and the modelling of version control. The process has been so divided for the sake of clarity, because of the different purposes of these two stages. The main model primarily aims at presenting the object types (forms) and their relationship, as well as attribute types. The relationship types represented in the main model are intended to provide the capabilities for checking consistency and traceability between forms. The modelling of version control, detailed in Section 6.3, shows the intrinsic and essential relationship types, which enable the proper associations between the different types of forms and their versions, created during subsequent system releases, to be established.

6.1.1 The COMFORM Model

The COMFORM model using the most important features of ORM is depicted in Figure 6.1. It illustrates the forms which are the phase products of the SMM, through the object, relationship and attribute types.

The object types in this model are the following SMM forms: Change Proposal, Change Approval, Maintenance Specification, Module Design and Module Source Code. The fields of the forms are either relationship or attribute types of the model. The relationship types link object types, providing a natural strategy for checking consistency and traceability between the forms. Every relationship type is followed by a minimum and maximum mapping, which characterizes the quantity of the relationship of a particular type in which the source object may be involved. The attribute types related to each object type (form) are the remaining fields of the forms, excluding the relationship types as described.

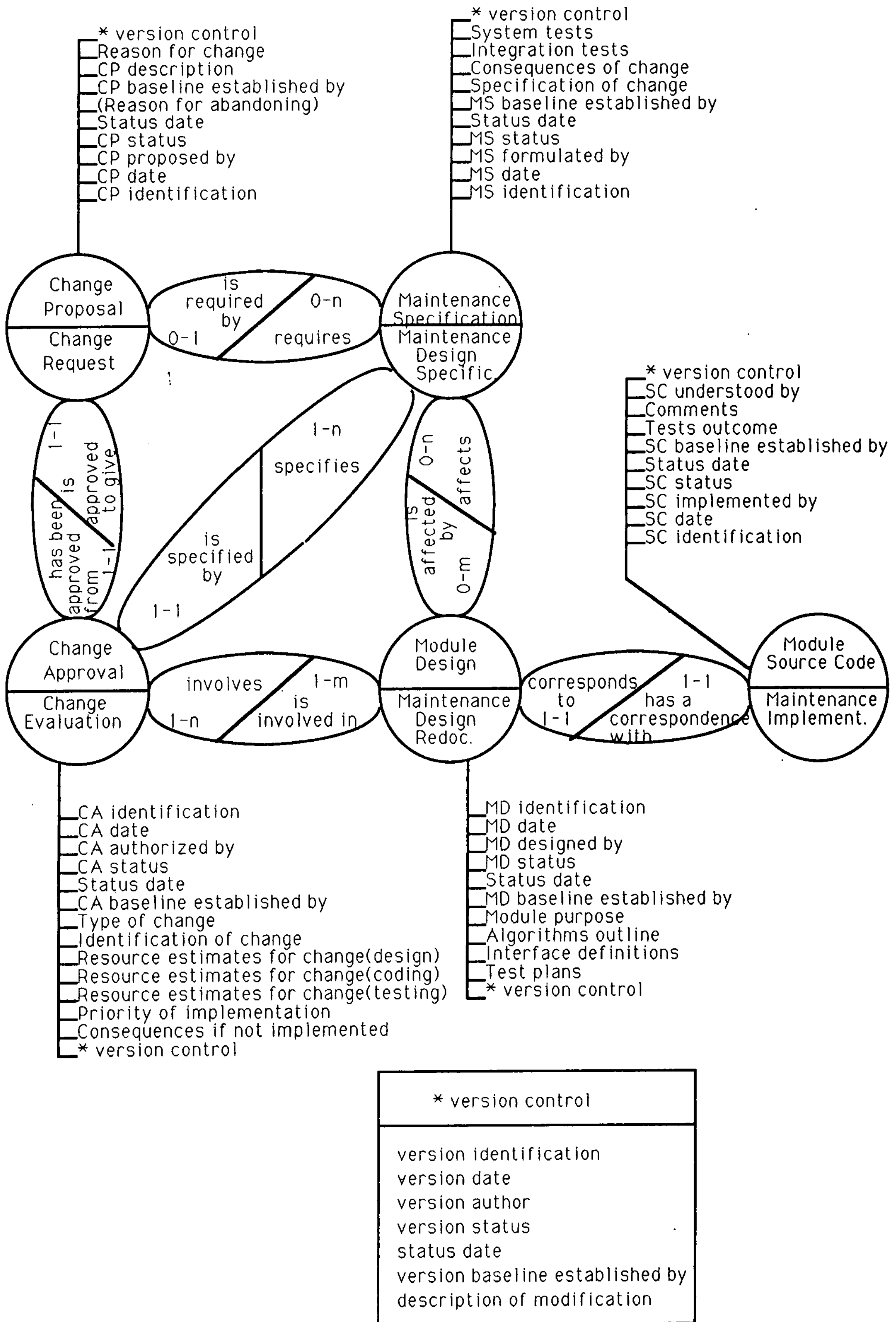

Figure 6.1: The COMFORM Model

6.1.2 Consistency and Traceability in COMFORM

The formalization of COMFORM can provide several benefits to help in the implementation of the method. The traceability and consistency of forms can be easily obtained by following the relationship types defined between them.

A Change Proposal form is traceable in the Change Approval form, through the relationship *is approved to give/has been approved from* which links them. This relationship will be generated after a proposed change is approved.

The Change Approval form is traceable in the Maintenance Specification form, through the relationship *is specified by/specifies* which exists between them. Since one specification can incorporate various approved changes, this relationship is represented in the Maintenance Specification form by the field *Related CAs.*

In order to establish the traceability of Module Design forms (in either Change Approval or Maintenance Specification forms), the fields *Involved* and *Affected sw components id.* represent the relationship types *involves/is involved in* between Change Approval and Module Design, and *affects/is affected by* between Maintenance Specification and Module Design forms respectively.

For every Module Source Code form, there must be a corresponding Module Design form and vice versa. In the model, this is represented by the relationship *corresponds to/has a correspondence with* which exists between them.

The relationship *requires/is required by* between a Maintenance Specification form and a Change Proposal form conveys the other changes (if any) which are necessary in order to fulfil the maintenance specification. The field *Other necessary changes*

in the Maintenance Specification form represents this relationship.

In COMFORM, a Change Proposal form characterizes the start of a maintenance activity and contains summary details of a proposed change in order to carry out its preliminary evaluation. Thus, the main evaluation information about a proposed change is contained in a Change Approval form. Consequently, the software components to be modified or created, (expressed in the model by the Module Design forms and Module Source Code forms), are associated mainly with the Change Approval form by the relationship *involves/is involved in.* The software components may also be associated with Maintenance Specification forms, since during the specification of the change, modifications to other software components may be required.

The Module Source Code forms are the nearest forms which link the software components to COMFORM. Their connection with the change specification, and the proposed change involving them, can be obtained indirectly through the relationship types defined in the model. There is a one-to-one relationship between a Module Source Code form and a Module Design form, which represents a higher level of software component documentation. On the one hand, a Module Source Code form is not directly associated with any other form than its corresponding Module Design form. The Module Design forms, on the other hand, may be associated with a Maintenance Specification form or a Change Approval form.

As far as mappings of relationships are concerned, there should be at least one Module Design form involved with a Change Approval form, though it may be that no Module Design form is affected by a Maintenance Specification form during the specification of the changes. Not all Module Design forms are necessarily involved with a Change Approval form or affected by a Maintenance Specification form.

However, a particular Change Approval form or Maintenance Specification form may involve or affect several Module Design forms at the same time.

A Configuration Release form may comprise several change proposals. For each approved Change Proposal form, there should be one and only one corresponding Change Approval form. All the approved change proposals of a particular system release should be specified in a single Maintenance Specification form. Because Change Approval forms are specified by a Maintenance Specification form in the model, the Change Proposal forms can be indirectly traced to their corresponding Maintenance Specification form.

Since it is not always the case that during the specification of a maintenance activity there will be *Other necessary changes* to be made, the mapping fields in this instance show that a Maintenance Specification form may require zero to several Change Proposal forms, and the Change Proposal form is required by only one (if any) Maintenance Specification form.

Constructing the relationship types as defined in this subsection allows one to navigate through all the associated forms in a particular system release. The current model has shown that not all object type need to be directly associated with each other to obtain the traceability and consistency checks among them. Moreover, the use of more relationship types would generate redundant information, which in turn could lead to more checks and increase the likelihood of generating inconsistencies.

It is worth drawing attention to the fact that the Configuration Release form is not represented in the main model of COMFORM. This is not due to the fact that such a form is irrelevant to the method, but because of its particular characteristic it requires no particular relationship type to link it with any other SMM form. The

importance of the Configuration Release form is unique in the model; it is detailed and discussed along with the modelling of version control in Section 6.3.

The cluster mechanism, detailed in the next section, provides further capabilities to aid COMFORM to define the scope of a system release, of a particular form and its versions, as well as positioning them in the whole structure defined by the method.

6.2 The Cluster Mechanism in COMFORM

The cluster mechanism is an additional concept of ORM, which encourages the grouping of forms in clusters, so that they can be organized in such a way that both the dependency and hierarchy can be established. This mechanism plays an important role in the modelling of COMFORM, since it provides the capabilities to implement various COMFORM concepts. The use of clusters enables the implementation of scheduled releases, and also facilitates the implementation of version control. In addition, the cluster mechanism allows the storage of information of several existing software systems in the same object base; it also helps in the task of establishing baselines, since every form belongs to its respective cluster, which in turn comprises all the necessary information for the task.

In the model of COMFORM, every cluster type comprises only one object type (form). This procedure affords a more precise version control, by which each form, rather than a set of forms, is controlled by the SCM discipline. The cluster types of the model are named after SMM phases. In addition, the object types they consist of are the products, i.e. the forms, which correspond to the result of these phases.

Although the cluster types represent different levels of abstraction, some of them may be at the same level in the hierarchy. The cluster hierarchy of COMFORM is depicted in Figure 6.2.

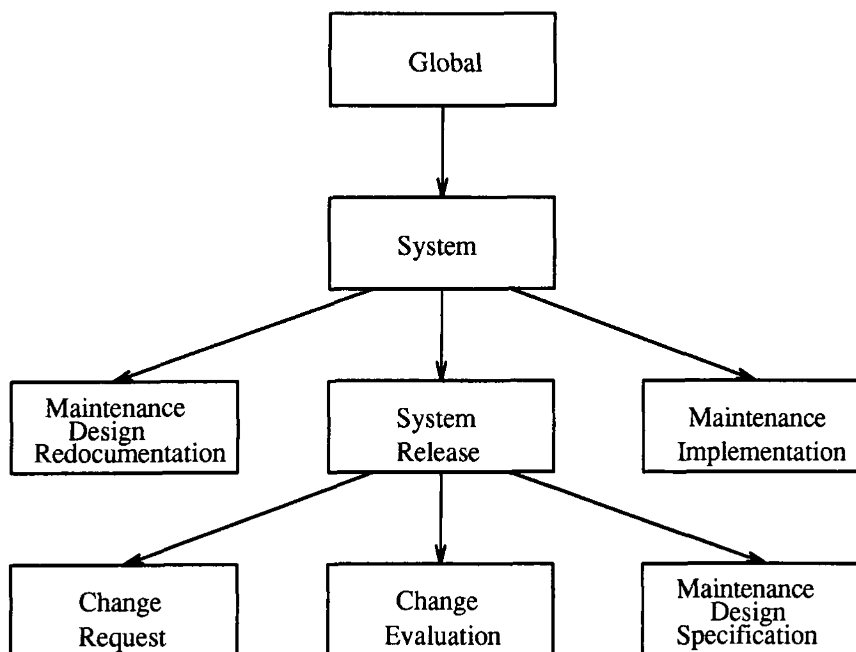

Figure 6.2: The Cluster Hierarchy of COMFORM

The *global* cluster type at the top of the hierarchy, is the most generic one. This cluster type is necessary to the enclosure of all other cluster types of the model. Tracing the hierarchy downwards, the next cluster type is named *system*, and envelopes all the information pertaining to a specific software system being maintained. In so doing, the model permits COMFORM to maintain several existing software systems, without the information in one interfering with that in another.

The third level in the hierarchy, comprises three cluster types at the same level named *system release*, *maintenance design redocumentation* and *maintenance implementation*. The *system release* cluster type comprises the information concerned with every system release. Therefore, it also includes the *change request*, *change evaluation* and *maintenance design specification* cluster types. On the same level

as the *system release* cluster type are the *maintenance design redocumentation* and *maintenance implementation* cluster types, which keep the system information of Module Design and Module Source Code forms respectively. As a result, when a cluster of objects in the *system* type becomes accessible, the maintainer can obtain further system information, which can be one of a particular Configuration Release, Module Design or Module Source Code form.

The *change request*, *change evaluation* and *maintenance design specification* cluster types contain the documentation of the maintenance required in a particular system release; therefore, they keep information about the Change Proposal, Change Approval and Maintenance Specification forms respectively. These three cluster types are under the *system release* cluster type, to enable the grouping of all forms documenting a particular system release in a single cluster.

Although the Module Design and Module Source Code forms are components of a Configuration Release, the *maintenance design redocumentation* and *maintenance implementation* cluster types are designed at the same hierarchical level as the *system release* cluster type. This is because Module Design and Module Source Code forms might be present in more than one Configuration Release form. Nevertheless, it is this characteristic which allows the software components be documented as they are being requested to undergo modification. Moreover, the incremental redocumentation in COMFORM is only possible because of the way the cluster hierarchy has been designed. In this way, software components are free to have their documentation improved at any time, since they are not part of any particular Configuration Release of the system. However, if they are requested to undergo modification in a particular Configuration Release, they will necessarily have to be documented.

6.2.1 The System Form

The level *system* has been created in the cluster hierarchy so that the implementation of the model can allow the storage of information about several existing software systems in the same object base. Following the model of COMFORM, (where every cluster type has an object type related to it), a System object type (form) is needed to keep the information concerned with every particular software system being maintained under COMFORM.

The System form depicted in Figure 6.3 presents the standard structure of SMM forms. It therefore contains the three sections: *identification*, *status* and *information*. While the *identification* and *status* sections include the standard information of forms, the *information* section includes all the Configuration Release, Module Design and Module Source Code forms that are part of a particular software system.

System

SR identification: Date:

SR originated by:

SR status: Status date:

SR baseline established by:

Comprises:

CRs:

MDs:

SCs:

Figure 6.3: The System Form

6.2.2 Scheduled Releases

One of the goals of using the cluster mechanism in COMFORM is the implementation of scheduled releases, and determination of the contents of each system release. The main advantage of scheduled releases is that changes are not introduced to the system at random. Change proposals should be organized and planned to be included in different system releases. This approach enables the optimization of maintenance tasks by grouping together those approved changes which involve the same software components, or have similar changes to be performed.

The design of the cluster hierarchy emphasizes the concept of scheduled releases. The *change request*, *change evaluation* and *maintenance design specification* cluster types are organized one level below the *system release* cluster type in such a way as to show that the Change Proposal, Change Approval and Maintenance Specification forms of a particular system release are part of a Configuration Release form. Therefore, all the Change Proposal forms associated with a particular system release must be fully developed in order to enable the release of a system configuration to the users. This means that the baseline associated with a Configuration Release form can only be established when all the forms associated with this form had their baselines already established.

6.3 Version Control

COMFORM aims to supply the capability of version control by providing a mechanism to manipulate versions of SMM forms, and to enforce restrictions on the

evolution of these forms, so that evolution is observable and controllable. To do so, the cluster mechanism concept of ORM has been employed. Within the model of COMFORM, each cluster type provides the structure to hold the information and all versions of a particular form, so that versions of a form are kept together and can be easily retrieved.

6.3.1 The Modelling of Version Control for COMFORM

Figure 6.4 depicts the relationship types between the object types (forms) included in a Configuration Release form. These relationship types are essential to the model in that they link the correct (version of) forms to their (version of) Configuration Release form. This figure also shows the relationship types which connect the Configuration Release, Module Design and Module Source Code forms to a particular System form. The attribute types associated with the Configuration Release and System forms are also shown in the same figure.

Since *maintenance design redocumentation* and *maintenance implementation* are not subcluster types of the *system release*, the model of version control requires that the relationship between them must be explicitly represented so as to associate the modified Module Design and Module Source Code forms with a Configuration Release form. Thus, the relationship type *is composed of/is a component of* represents the link between the Configuration Release and Module Design, and between the Configuration Release and Module Source Code forms.

The relationship type *comprises/is part of* identifies the forms which are part of a cluster type. As the *change request*, *change evaluation* and *maintenance design spec-*

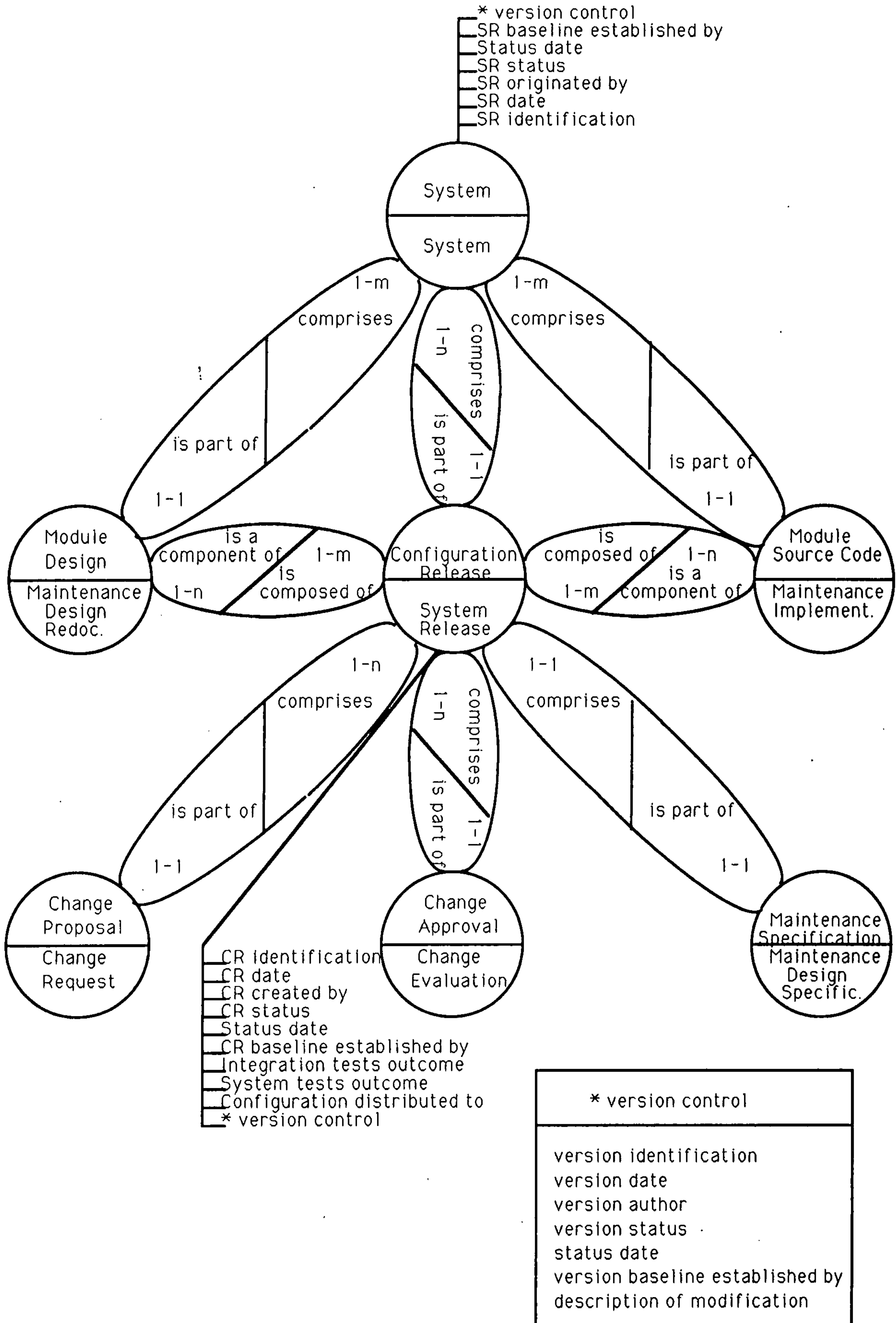

Figure 6.4: The Modelling of Version Control for COMFORM

ification are subcluster types of the *system release* cluster type, the Configuration Release form is associated with Change Proposal, Change Approval and Maintenance Specification forms through the relationship type *comprises/is part of.* This relationship type is not an explicit one (since it is part of the cluster mechanism concept), but is represented in the model in order to clearly show that the Change Proposal, Change Approval and Maintenance Specification forms are part of a particular Configuration Release form.

In the same way as the *system release*, *maintenance design redocumentation* and *maintenance implementation* are sub-cluster types of the *system* cluster type, the System form is associated with Configuration Release, Module Design and Module Source Code forms by the relationship type *comprises/is part of.*

As far as mappings of relationship types are concerned, a Configuration Release form may comprise one or more Change Proposal and Change Approval forms, but only one Maintenance Specification form which specifies all the changes approved for that particular system release. On the other hand, Change Proposal, Change Approval and Maintenance Specification forms are part of only one Configuration Release. The software components (represented by the Module Design and Module Source Code forms) which have been involved in an approved change, or affected by a maintenance specification of a particular Configuration Release, are the components of a Configuration Release form. Therefore, there should be at least one Module Design and Module Source Code form associated with a Configuration Release form.

The existing software system being maintained using COMFORM comprises at least one Module Design, Module Source Code and Configuration Release form, since there is at least one software component to be maintained, which in turn has to be

developed by generating a Configuration Release form. On the other hand, every Module Design, Module Source Code and Configuration Release form is part of only one System form.

6.3.2 The Modelling of Revisions and Variations

The generation of versions from a particular form demands the creation of specific relationship types to control them. Therefore, the relationship types, *is a revision of/generates revision* and *is a variation of/generates variation*, between an original and a versioned object type represent the concepts of revision and variation for the method (described in Section 5.4). This is depicted in Figure 6.5.

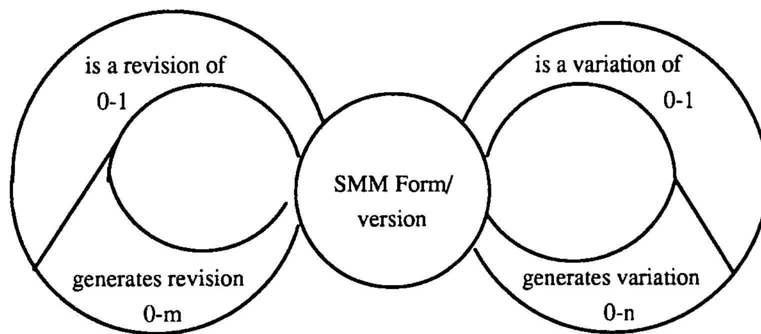

Figure 6.5: The Modelling of Revisions and Variations

The relationship types, *is a variation of/generates variation* and *is a revision of/generates revision*, provide the mechanism to follow all the versions of a particular form throughout its existence, by providing the history as well as the current status of its evolution. These relationship types are created within a cluster of objects, and link two versions of the same object.

The relationship type, *is a variation of/generates variation* implements the concept

of variation, allowing the visualization of the sequence in which variations of a form have been created.

The relationship type, *is a revision of/generates revision*, shows the sequence in which revisions of a particular form have been created. This relationship type allows the history associated with each form of an existing software system to be obtained.

A form or its version may generate zero to several revisions or variations; however, a version (revision or variation) is generated by only one (if any) form or version.

6.4 An Illustrative Example

In order to clarify the COMFORM model described in this chapter, consider a simple example given in Figure 6.6. This figure shows the clusters of objects (ovals) and forms (rectangles) with their versions, and illustrates the notion of cluster hierarchy. For the sake of simplicity, the name of the cluster types will be omitted, since each SMM form is associated with its respective cluster type named after the SMM phase. The different cluster types are illustrated in the figure by the different patterns of the ovals.

System1 is the form associated with the cluster type *system*, representing an existing software system being maintained by COMFORM. In this example, *System1* has two clusters of objects of the *system release* type depicted by the forms *CR1* and *CR2*. *CR2* is being developed so that it contains one form in each of the cluster of objects enveloped by it. A Change Proposal form *CP2* has been approved and has a Change Approval form *CA2* associated with it. The approved change is specified in a

Maintenance Specification form *MS2*. Furthermore, the approved change modifies a software component whose Module Design and Module Source Code forms are *MD3* and *SC3* respectively. These forms are components of the Configuration Release form *CR2*, whose relationships are depicted in that figure by arrows.

The Configuration Release form *CR1* has already been developed. In its original version, a Change Proposal form *CP1* was proposed and abandoned. This same Change Proposal was later approved (*CP1-1.1*) and its modification carried out. The Change Approval form *CA1* was created in response to the approval of the change. The Maintenance Specification form *MS1* contains the specification of the approved change. The *CP1-1.1* has proposed a modification to an existing software component, whose Module Design and Module Source Code forms are named *MD1* and *SC1* respectively. As a result of the proposed change, a version of these forms has been created (*MD1-1.1* and *SC1-1.1*) and associated with the Configuration Release form *CR1*.

However, the proposed change (*CP1-1.1*) was not carried out properly. Once the forms associated with *CR1* had been frozen by the SCM control, they could no longer be modified, so that a version of the Configuration Release form *CR1* had to be created (*CR1-1.1*) in order to amend it. As a result, a new Change Proposal form has filled in (*CP1'*) so as to have the problem properly documented. Its approval created the Change Approval form *CA1'*. The specification of the change was documented in the Maintenance Specification form *MS1'*. The latest version (1.1) of the software component (whose Module Design and Module Source Code forms are the *MD1-1.1* and *SC1-1.1* respectively) were then modified and the corresponding versions (*MD1-1.2* and *SC1-1.2*) of the forms were created. The specification of the change, however, also required modifications to another software component, represented by

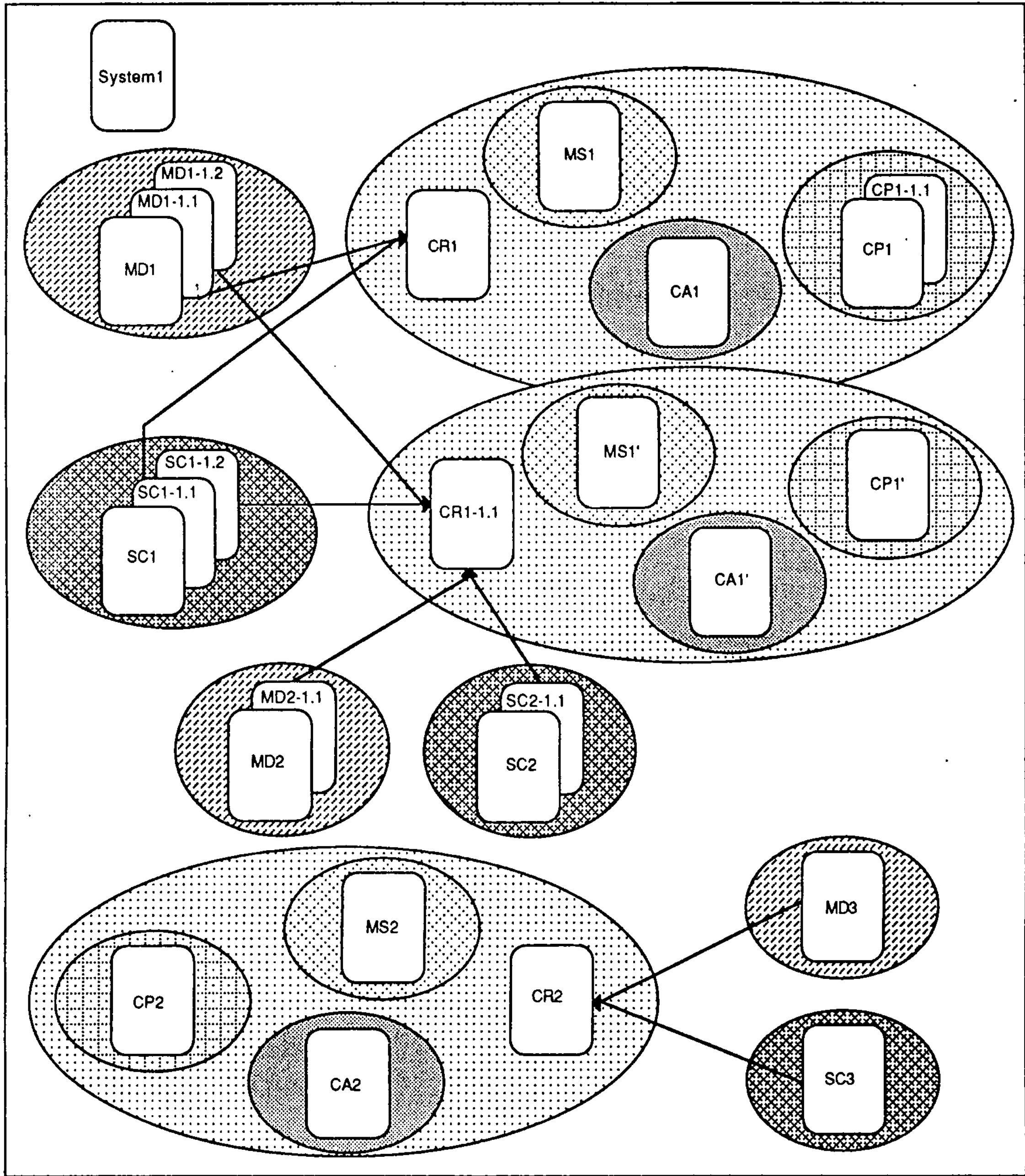

Figure 6.6: An Illustrative Example of the Modelling

the forms *MD2* and *SC2*, which in turn led to the creation of new versions (*MD2-1.1* and *SC2-1.1* respectively).

It is clear from this example that the information for a particular software system being maintained by COMFORM can be kept separate in an object base. Once a particular software system is instantiated, information about its system releases and software components (Module Design and Module Source Code forms) can be obtained. If details of a particular system release are required, the cluster of objects which comprises its corresponding Configuration Release form provide all the required information.

6.5 Comments on the Formalization of COMFORM

The formalization of COMFORM has been presented in this chapter using the basic concepts of ORM. The model covers the basic features of the method proposed in Chapter 5.

The current model has not dealt with aspects of software project management. Therefore, people involved in the whole process (maintainers, managers and users) have not been represented as object types in order that they might be given closer attention as software configuration items. This has not been developed in the current model, since it was not the objective of this research to perform any sort of statistics or metrics involving information about each of the maintenance team members.

The current hierarchy of clusters has been designed in such a way that each cluster type comprises all versions of a specific form. In addition, the cluster hierarchy allows the division of forms into distinct hierarchical levels, so that traceability between forms can be obtained by distinguishing the levels that the maintenance process goes through. Therefore, this hierarchy shows how the different types of forms interact with each other. Since the *system release* cluster type is the focus of a maintenance process, it is this cluster which envelopes the cluster types of *change request*, *change evaluation* and *maintenance design specification*. The *maintenance design redocumentation* and *maintenance implementation* cluster types are at the same level as the *system release*, since the Module Design and Module Source Code forms do not belong to any specific system release.

Finally, the formalization of COMFORM is an important step towards the implementation of an automated support, required for the method. The implementation of a framework capable of supporting such a model provides users with a better understanding of the method. In addition, the quality assurance checks can be automated so that a more reliable and robust software maintenance environment can be obtained. The next chapter discusses the details of the implementation carried out in this research in order to automate COMFORM.

Chapter 7

The COMFORM Prototype

The previous chapters described the method underlying COMFORM and its formalization. This chapter discusses some important issues encountered when developing a prototype for COMFORM. The prototype forms the foundation of a software maintenance environment, into which are incorporated the capabilities and ideas that have been investigated, in order to create COMFORM. The prototype focuses on the implementation of the main concepts introduced by COMFORM.

7.1 Implementation Issues

The main purpose of the prototype is to provide an automated framework into which other tools can be incorporated, so that this framework represents the ba-

sis of a software maintenance environment, supporting the method described in Chapter 5. Hence, the prototype aims to provide automated support for change request, change release, version control, configuration management and incremental redocumentation. A major feature of the prototype is that maintenance and system information is stored in a unified object base, making it easy for people involved in the maintenance process (maintainers, users and managers) to query and analyse the software system being maintained.

The implementation of this prototype involves the COMFORM kernel (which represents automation of the model described in Chapter 6), the COMFORM framework (which allows the incorporation of other tools), and a Report Generator tool. The prototype was implemented in the Unix environment using the C programming language [65] and consisted of about 8000 lines of C source. The COMFORM prototype enforces the characteristics of the method, such as completeness, consistency, traceability and version control.

The following sections of this chapter describe the COMFORM kernel, the COMFORM framework and its operations, the report generator, the implementation of the basic features of version control in greater detail, and discuss ways to integrate COMFORM into the Unix environment.

7.2 The COMFORM Kernel

The COMFORM kernel consists of a set of primitives which store, recover and update the system information that semantically represents the SMM forms and

their fields. These primitives manipulate objects, relationships and attributes of the modelling described in Chapter 6.

The primitives of the COMFORM kernel are at the lowest level of implementation, representing the uniform mechanism with which to manipulate the COMFORM object base, assuring its integrity. Such primitives not only perform the required tasks, but also perform consistency and integrity checks on data being added to the object base. For instance, if a request is made to establish a relationship by calling the insert relationship primitive, the COMFORM kernel first checks if that relationship is allowed between the requested object types. If valid, a check is then made to see if the link is redundant, and if not, a new entry is automatically inserted in the object base for the established relationship.

The primitives that have been implemented take into account the cluster hierarchy, as described in Chapter 6. The forms can be grouped in clusters so that they are organized in such a way that dependency, as well as levels of hierarchy among them can be established.

The COMFORM kernel also provides specific primitives for the version control, in order to keep track of modifications to the forms. Details of the version control approach taken by the prototype are discussed later in Section 7.5.

7.3 The COMFORM Framework

In order to automate the method effectively, tools should be implemented and added to the COMFORM framework. This framework is the primary means by which SMM

forms are made to interact with people involved in the maintenance process. The implementation of tools involves calling several primitives from the COMFORM kernel, in order to manipulate the information in the object base.

The COMFORM framework is the main part of the prototype, representing the basis of a software maintenance environment, where other specific-purpose tools can be added. In the prototype, it consists of operations to manipulate the Change Proposal, Change Approval, Maintenance Specification, Module Design, Module Source Code, Configuration Release and System forms. The fields of the forms are interpreted according to their specific functions, such as attribute and relationship types of the model, described in Chapter 6.

The following operations manipulate the forms in the COMFORM framework: *create, get, modify, delete, effectuate, establish baseline* and *undo effectuate.* The operations have been designed in such a way that they automate the important basic functions of COMFORM. The operations of the COMFORM framework are described in more detail in the following subsections. Such operations manipulate either the forms or the alternatives of these forms (but for the sake of simplicity, only the forms are referred to in the description of the operations).

7.3.1 The CREATE operation

The *create* operation allows the user to create any SMM form. Forms can only be created if they comply with the cluster hierarchy as defined during the formalization of the method. In particular, a Change Approval form is only created if there is at least one Change Proposal form whose status is *effective*, so that the new form

can be associated with a specific and approved Change Proposal form. Change Proposal forms, Change Approval forms and Maintenance Specification forms can only be created in a *system release* cluster of objects whose status is *in development*. Furthermore, a Configuration Release form has to be created first for each new system release.

7.3.2 The GET operation

The *get* operation displays the information of a specific form. The three sections of the forms (*identification*, *status* and *information*) are displayed. In addition, if a form contains alternatives, they are all displayed along with their current status. System and Configuration Release forms also displays those other forms which are part of them, thus providing the information contained from that level and downwards in the cluster hierarchy. Therefore, a System form displays the Configuration Release, Module Design and Module Source Code forms, which are part of the cluster of objects associated with that particular form; and a Configuration Release form displays the Change Proposal, Change Approval and Maintenance Specification forms, which are part of the cluster of objects associated with that particular form. A Configuration Release form also displays the Module Design and Module Source Code forms, which are components of that particular form.

7.3.3 The MODIFY operation

The *modify* operation allows the user to modify or fill in the fields of any existing form whose current status is *in development.*

7.3.4 The DELETE operation

The *delete* operation removes a specified form from the COMFORM object base. Initially, a check to determine whether or not the specified form is *in development* is carried out.

7.3.5 The EFFECTUATE operation

The *effectuate* operation is responsible for checking the completeness of a form. This operation brings the specified form to an intermediate status so that it cannot be modified or deleted. The specified form has to be *in development* and all of its fields have to be filled in. The only exception to this rule is the Maintenance Specification form, where the *Other necessary changes* field does not always have to be filled in to meet a specification. The intermediate status (*effective*) is the first step towards the establishment of a form baseline.

7.3.6 The ESTABLISH BASELINE operation

The *establish baseline* operation is the point at consistency and traceability checks are performed. This operation represents the next step, after the *effectuate* operation has been completed. Here, checks are performed not on the forms individually, but on the set of forms which are part of the cluster of objects associated with the form whose baseline is to be established. This operation changes the form status from *effective*, to the permanent *frozen* status. The necessity for following a sequence in this operation, and the reasons for it, have already been detailed in Chapter 5.

The lowest stage of this process is represented by the Module Source Code forms, where the only requirement for establishment of a baseline is that the specified form has one, and only one *has a correspondence with* relationship with a Module Design form, whose status is *effective.*

The next stage is the establishment of baselines of the Module Design forms, since they are linked and have a one-to-one relationship with the Module Source Code forms. Since the Module Design forms and Module Source Code forms are not part of a particular system release, because they are part of a system only, they can be treated independently. Consequently, these forms can be documented at any time, thus adding to the system documentation. These two types of forms can also have their baselines established at any time, as long as they have their correct corresponding relationships. There must be only one relationship between a Module Design form and a Module Source Code form. Nevertheless, these forms do have to be fully documented if they are components of a system configuration to be released.

As far as the documentation of the maintenance process is concerned, the Change

Proposal form is the first one to have its baseline established by approving or rejecting a proposed change. The Change Approval forms and the Maintenance Specification form are the next ones, as long as the software components (Module Design and Module Source Code forms) involved in and affected by those forms have already had their baselines established. The implementation of modification having been carried out already, the next step involves the final evaluation of that maintenance process, which requires the filling in of the Configuration Release form in order to have its baseline established.

7.3.7 The UNDO EFFECTUATE operation

The *undo effectuate* operation acts as a safety valve, bringing an *effective* form back to *in development* status, so that modifications can again be carried out on that form. The situation of having to return to *in development* status is sometimes required, since, during the establishment of baselines of forms, there may be some inconsistencies which can only be amended if modifications to some fields of forms are performed. For instance, if during the implementation of a Change Proposal the development of a new software component is required, such information has to be documented in the corresponding Change Approval form. If it is the case that this Change Approval form has already been effectuated, the *undo effectuate* operation has to be performed in order to establish the baselines of the forms related to this change.

7.4 The Report Generator Tool

The Report Generator tool plays a key role in the COMFORM prototype as far as the SCM function of Software Configuration Status Accounting is concerned. It is this tool which retrieves the information stored in the object base, and presents it in its various ways to those people involved in the maintenance process (maintainers, users and managers).

Different types of reports can be generated by choosing parameters and options which represent the information that is required to be displayed. In so doing, the prototype generates either managerial or technical reports, such as those described in Subsection 5.3.3.

Table 7.1 depicts the parameters and options of the report generator. The user can define particular reports, by choosing particular parameters and options:

1. Whether or not the name of the current cluster is to be displayed.

2. Whether or not the form totals in a report are to be displayed.

3. Information to be displayed in a report can be selected from:

    - form name
    - creation date
    - author of creation
    - current status
    - status date
    - author of baseline.

1. Name of current cluster				display (Y/N)	
2. Total of forms				display (Y/N)	
3. Information to be displayed					
3.1 Form name		3.2 Creation date		3.3 Author of creation	
3.4 Current status		3.5 Status date		3.6 Author of baseline	
4. Type of forms to be displayed					
4.1 Configuration Release		4.2 Module Design		4.3 Module Source Code	
4.4 Maintenance Specification		4.5 Change Approval		4.6 Change Proposal	
5. Optional parameters to be matched					
5.1 Form name		5.2 Creation date		5.3 Author of creation	
5.4 Current status		5.5 Status date		5.6 Author of baseline	

Table 7.1: Parameters and options of the report generator

4. Types of forms to be displayed in a particular report. For instance, the Change Proposal forms of an existing software system.

5. Optional parameters to be matched, so that forms with specific characteristics can be displayed. Such an option allows, for instance, the generation of a report containing the forms whose status is *in development* or the forms created by a particular author.

7.5 Version Control

A simple form of version control has been implemented in the COMFORM prototype. All types of SMM forms can be brought under version control, although the main use of versions are for the Module Design and Module Source Code forms.

Alternatives of any form can be created, as long as the form has been *frozen*. Additionally, alternatives are only allowed to be added to *in development* clusters of objects. Alternatives are identified by the identifier of the original form, and by

a version number. For instance, an alternative for the *ms1* form could be named *ms1-1.1*, representing a further step in the development of the *ms1* form.

In the current prototype (just to illustrate this concept), a simple structure has been implemented so that alternatives are always generated from the original form. A more reasonable way of generating alternatives would be from the last version of a form, or by allowing the maintainer to choose from which version of a form the alternative should be generated. Thus, to generate an alternative, the content of fields of the original form are copied. In addition, a new field, *Description of modification*, has been introduced, where the reasons for generating a new alternative must be documented. The other fields of the form may remain the same, if changes to them are not required. However, the fields which are represented by relationships in the model are not copied from the original form, making the maintainer check if those forms related to the one being modified, require the generation of new versions or not. For instance, in the Module Source Code form, the field *Corresponding MD* of which the information required is the identifier of the corresponding Module Design form, is not duplicated from the original form.

All checks performed on an original form are also performed on alternatives, so that to effectuate an alternative, all of its fields must be filled in. Furthermore, to have its baseline established, an alternative must comply with the same requirements applied to the original form. Moreover, alternatives are treated as independent forms when they are referred to in the fields of other forms. For instance, in a Change Approval form and Maintenance Specification form, the names of the Module Design forms are required in the fields *Involved sw components id.* and *Affected sw components id.* respectively. If the name of an alternative were to be given, only when that particular alternative has had its baseline established (becomes a version) could

those former forms have their baselines established.

7.6 COMFORM in the Unix Environment

Since there are already established tools in the Unix environment to control source code, (for example RCS [108, 109] and *make* [39]), the main purpose of COMFORM has been to generate incremental documentation to existing software systems. Hence, COMFORM does not control the software components directly. Instead, a link can be established with such tools, extending COMFORM's capabilities to ensure that not only the documentation generated by COMFORM, but also the source code being maintained by RCS and *make* are consistent and under control.

In the current implementation of the COMFORM prototype, the tools RCS and *make* have been used. The link established between COMFORM and these tools are such that the components of existing software systems can be maintained using RCS to keep their versions under control, and *make* to assure that the executable code is being generated using the updated versions of the software system.

To provide the link between COMFORM and other tools, a program called *checker* has been written. The *checker* program looks at RCS and *make* information and converts it into the format expected by COMFORM. The first step towards having these Unix tools working together with COMFORM is the introduction of the software components of an existing software system to RCS, using its *ci* and *co* commands. At the same time, a *makefile* file should be created in order to give the dependency information about the existing system to the *make* utility. From the in-

formation contained in the *makefile* file, the *checker* program is able to recognize the software components of the existing system and introduce them to the COMFORM object base.

Since the introduction of an existing software system to the RCS tool generates the revision 1.1 to all software components, COMFORM also accepts the first alternative of all introduced software components as 1.1. Therefore, the *Description of modification* field of the corresponding Module Source Code form is *Initial revision* as automatically created by the RCS tool. Any other new revision created by using RCS can be detected by the *checker* program. Additionally, the RCS log message inserted during the creation of that revision can be automatically transferred to COMFORM as the *Description of the modification* field of the corresponding Module Source Code form.

The way that RCS/*make* and COMFORM can be kept in step is by always running the *checker* program before releasing a new system configuration, i.e., before the establishment of baselines process. The *checker* program is then able to point out the new software components and the new versions of existing software components still not included into the COMFORM object base. These new software components are then automatically inserted to the COMFORM object base. Only after the successful running of the checker program can the baselines of the software system be established and, therefore, have a new configuration released to the users.

To sum up, the *checker* program is vital in the process of linking Unix tools and COMFORM. At the introduction of the existing software system to COMFORM, the role of this program is to transfer all the software components inserted in the *makefile* file to the COMFORM object base. During the maintenance process of

the introduced software system, the *checker* should be run before establishing the baselines (to release a system configuration), so that all the new software components and new versions of existing software components can be detected and transferred to the COMFORM object base.

7.7 Summary

Although the implementation of COMFORM described in this chapter does not represent a software maintenance environment on its own, the prototype contains the essential characteristics of the method as discussed in Chapter 5. The prototype has been shown to be capable of performing such fundamental quality assurance checks as completeness, consistency and traceability while incrementally recovering the documentation of software systems, consequently improving their maintainability. Moreover, the COMFORM kernel, as the only interface to access the information from the object base, ensures the integrity of the data.

The completeness of forms is obtained through the *effectuate* operation by upgrading the forms' status only if all their fields are filled in. The traceability check is performed during the *establish baseline* operation. During this operation, the set of forms that is part of a cluster of objects associated with a Configuration Release form is traced, in order to ensure that those forms are in agreement with each other.

Basic consistency checks have also been implemented in the COMFORM prototype as described below:

- Change Approval forms are only created from Change Proposal forms that

have just been approved.

- The redocumentation of a software component consists of completing the two correlated forms, Module Design and Module Source Code.

- A system release can only be delivered to users if all the change proposals associated with it are sorted out, i.e., the change proposals have been implemented.

- A Maintenance Specification form and Change Approval forms can only have their baselines established if the affected and involved software components have been documented, and have therefore had their baselines established.

The COMFORM prototype has been developed in order to provide the automated support necessary to evaluate the underlying method. Such a support facilitates project management, status reporting, auditing and quality assurance. It can also be easily used to control and track changes and forms being manipulated. As a result, the COMFORM prototype is able to provide a complete audit trail from change proposal to change release. In addition, COMFORM improves change control, because personnel are unable to introduce a change to a software system without a valid approved change proposal.

The next chapter describes an example of COMFORM use, in order to illustrate its applicability and to clarify the points of implementation described in this chapter.

Chapter 8

Use of the COMFORM Prototype

This chapter describes the use of the COMFORM prototype for maintenance of an existing software system. The first section describes the steps that should be followed during COMFORM use. Section 2 details the application of COMFORM to an existing software system called PXR, which is a context-sensitive cross-reference tool. In the last section, general comments and conclusions are made.

8.1 The COMFORM Steps

The COMFORM prototype supports the method for the maintenance process, as presented in Chapter 5. Hence, the way of using the COMFORM prototype is in accordance with the guidelines and procedures established by that method. Although

the level of complexity may vary according to the type of maintenance being performed in existing software systems, the most important steps of the COMFORM prototype are as follows.

Step 1. Create the COMFORM object base.

This step should be performed only once to create the files which comprise the COMFORM object base. This object base will store all information related to the maintenance of any existing software system supported by COMFORM.

Step 2. Introduce an existing system to COMFORM.

Since the method requires no previous complete documentation of an existing software system in order to start operating, the Software Configuration Identification function must be the first step taken when using COMFORM. In fact, the only knowledge of existing software systems required by COMFORM is of their software components. For each software component, there will be a corresponding Module Source Code form in the COMFORM object base. In this way, the software components will be known to COMFORM. As a software system is being maintained, the knowledge obtained during the understanding and modifications of the software components is kept. Consequently, only modified software components are compulsorily documented before a new system configuration (using those software components) is released.

Step 3. Introduce change proposals to COMFORM.

On reaching this step, COMFORM is ready to act in the maintenance process of the introduced existing software system. This step is triggered by change proposals which should be grouped into configuration releases, so that the maintenance process can be scheduled and improved.

Step 4. Evaluate change proposals.

The proposed changes are evaluated by the maintenance staff and may be approved or rejected. The automatic procedure for the approval of change proposals triggers the creation of their corresponding Change Approval forms. The approved changes are further reviewed by a Change Control Board, and are then classified and prioritized. There will be a reason for the rejection of certain change proposals, which is explained and kept in the COMFORM object base, adding to software system history.

Step 5. Document the maintenance process itself.

During the evaluation of the approved changes, the Change Approval forms are filled in. Correspondingly, the Maintenance Specification form is also filled in, along with the specification of all the selected approved changes for a particular system release. The maintenance process is documented and the system evolution history is kept in the COMFORM object base.

Step 6. Document the software components to be modified.

During evaluation and documentation of the approved changes, those software components that need to be modified will emerge. Therefore, as the Maintenance Specification form and Change Approval forms are filled in, the information about those software components is discovered and kept by filling in their corresponding Module Design and Module Source Code forms.

Step 7. Implement and test the changes on the software components.

At this point, the required modifications to the software components should be implemented. After that, the individual tests planned for each software component (documented in the Module Design form) should be carried out, and the results recorded in the corresponding Module Source Code form. After the successful completion of individual tests, the integration and system tests should be carried out, and the results recorded in the Configuration Release form.

Step 8. Initial evaluation of forms.

Once the changes have been implemented on the affected and involved software components and the tests have been carried out successfully, all forms of the system release should have their baselines established, in order to have them released to the users. The initial evaluation of the auditing process includes checking the completeness of the forms, i.e., the forms should be fully completed, in order to have their baselines established. The *checker* program should be run to ensure that the modi-

fications performed in the source code are updated in the COMFORM object base. This program detects all versions of the existing software components and the new software components that have been created but not updated in the COMFORM object base.

Step 9. Establish the baselines of forms.

The function of establishing baselines is an important step within the method, since it is at this point that the quality of the software system being maintained is assured. The successful outcome of the integration and system tests (performed at the end of the implementation of the required modifications) is transcribed into the Configuration Release form, enabling the corresponding system configuration to be released. Establishment of baselines is performed after the initial evaluation of the auditing process, and involves checking the consistency of the forms, (i.e., the correlated forms must be in accordance with each other), and also checking the traceability between the different types of forms. Such a process starts at the lowest level, i.e., at the Module Source Code forms. After the Module Source Code baseline has been established, the corresponding Module Design baseline should be established. At this stage, the Change Approval baseline of this configuration release should also be established. After that, the Maintenance Specification baseline can be established.

Step 10. Release the system configuration to the users.

Once the baselines of all the forms of the system release have been established, the baseline of the Configuration Release can be established and made available to the users.

Step 11. Generate reports.

A number of reports can be obtained to provide answers to a variety of queries about the software systems kept under COMFORM. Reports such as those listed in Subsection 5.3.3 help the people involved in the maintenance process to perform their tasks.

8.2 Example of COMFORM Use

Although the COMFORM object base accepts the storage of information of more than one existing software system, the example of COMFORM use described here deals with only one software system. The COMFORM prototype has been applied to the maintenance of a context-sensitive cross-reference tool named PXR [19], developed at the University of Durham. This tool is an 8000 line 29 module program, written in Pascal. Although the size of this software system is not large compared with the majority of existing software systems being maintained in industry, it is believed to be of sufficient size for the evaluation of the COMFORM prototype's performance in maintaining existing software systems. Its simple functions, as well as its compact size, allow a clearer illustration of the concepts of COMFORM (presented in Chapter 5) to be made.

The PXR system produces two types of cross-reference listings of Pascal programs. The first one, called block-structured cross-reference, is a structured cross-reference listing, in which each identifier occurring within a program block is listed alphabetically within that program block. The second one is more like the conventional

cross-reference listing, producing an alphabetical listing of identifiers indicating region, line declared, class of object represented and the lines on which the identifier is used in each way.

The rest of this section describes the employment of COMFORM to maintain the PXR system, following the steps described in the previous section.

Step 1. Create the COMFORM object base.

After creating the COMFORM object base, the name of the software system is added. Figure 8.1 shows the initial information put in the COMFORM object base,

```
-----------------------------------------------------------------
SR identification: PXR     date: 28/01/92
SR originated by:  des3mmc
-----------------------------------------------------------------
SR status: in_development      status date: 28/01/92
-----------------------------------------------------------------
           Comprises:
CRs:
MDs:
SCs:
-----------------------------------------------------------------
```

Figure 8.1: Initial information in the COMFORM object base

which consists of the name of the software system, the date and the author of its creation. Since the PXR system has just been introduced to the COMFORM object base, its current status is *in development* and Configuration Release, Module Design, and Module Source Code forms have not yet been created to document the system.

Step 2. Introduce an existing system to COMFORM.

In the implementation of COMFORM carried out for this thesis, the source code of the existing software system is assumed to be maintained using the RCS tool, together with *make*. At first, COMFORM must know the names of the existing software components being introduced. In the COMFORM prototype, a software component is assumed to be a single file. COMFORM makes use of the *makefile* information to bring all the existing software components under COMFORM, thus performing the Software Configuration Identification function of the SCM discipline.

Figure 8.2 shows the System form containing the Module Source Code forms associated with the PXR software components, which have been introduced into the COMFORM object base. The insertion of these Module Source Code forms is achieved by the *checker* program, which links all the software components from the *makefile* file with a corresponding Module Source Code form in the COMFORM object base.

Step 3. Introduce change proposals to COMFORM.

In order to illustrate the use of COMFORM, five change proposals were elaborated so as to introduce changes into the PXR system. Most of these changes were enhancements (perfective maintenance), whose implementation priorities were *important* or *minor changes*. The proposed changes could thus be organized and planned to be implemented in different scheduled system releases. This approach was taken in order to improve the maintenance process, by grouping together those change proposals which involved the same software components, or had similar changes to be performed.

```
-----------------------------------------------------------------
SR identification: PXR     date: 28/01/92
SR originated by:  des3mmc
-----------------------------------------------------------------
SR status: in_development      status date: 28/01/92
-----------------------------------------------------------------
          Comprises:
CRs:
MDs:
SCs:
     arguments.p
     error.p
     expression.p
     getconstant.p
     gettoken.i
     gettoken.p
     gettype.p
     idstack.p
     includestack.p
     linenumber.p
     literal.p
     paramlist.p
     print.p
     pxr.p
     readblock.p
     readconst.p
     readfile.i
     readfile.p
     readlabel.p
     readprogram.p
     readroutine.p
     readtype.p
     readvar.p
     statement.p
     stripzeroes.p
     symbol.i
     symbol.p
     symbolstack.p
     variable.p
-----------------------------------------------------------------
```

Figure 8.2: Software components in the COMFORM object base

Consequently, two Configuration Release forms have been created. The first one (*pxr-1.1*) containing two Change Proposal forms (*cp-1* and *cp-2*), and the second one (*pxr-1.2*) containing three Change Proposal forms (*cp-3, cp-4* and *cp-5*). The first two change proposals have been grouped in *pxr-1.1* because they are modifying the same software component (*arguments.p*). In the second system release, *cp-3, cp-4* and *cp-5* are together because they do not require immediate attention. For simplicity, in this example only *pxr-1.1* has been further developed. Figure 8.3 shows the *cp-1* form of *pxr-1.1*, as an example of a Change Proposal form.

```
---------------------------------------------------------------------
CP identification: cp-1     date: 28/01/92
CP proposed by: des3mmc
---------------------------------------------------------------------
CP status: frozen      status date: 28/01/92
CP baseline established by: des3mmc
---------------------------------------------------------------------
1. CP description: To parse f option
2. Reason for change: Extend to allow full alphabetic and
   structured listings
---------------------------------------------------------------------
```

Figure 8.3: A Change Proposal form

Step 4. Evaluate change proposals.

These proposed changes have been evaluated and approved. The approval of the changes automatically generated the corresponding Change Approval forms. Figure 8.4 shows the change approval *ca_cp-1*, after it has been completed. The field *Related CP* ties the form *ca_cp-1* with its corresponding form *cp-1*. The analysis and review of *cp-1* identified the proposed change as being perfective maintenance, its work being recorded in the field *Identification of change*. The impact analysis

```
------------------------------------------------------------------
CA identification: ca_cp-1     date: 28/01/92
CA authorized by:  des3mmc
------------------------------------------------------------------
CA status: in_development      status date: 28/01/92
------------------------------------------------------------------
1. Related CP:
         cp-1
2. Type of change: perfective
3. Identification of change: Modify procedures Usage and
   Three_char_param.
4. Involved sw components id.:
         arguments-design
5. Resource estimates for change (design): 1 hour
6. Resource estimates for change (coding): 15 minutes
7. Resource estimates for change (testing): 30 minutes
8. Priority of implementation: important
9. Consequences if not implemented: Options not available
------------------------------------------------------------------
```

Figure 8.4: A Change Approval form

carried out shows that the software component *arguments.p* requires modification. Such information is shown in the form, where the field *Involved sw components id.* records the name of the corresponding Module Design form (*arguments-design*). The resources estimates for designing, coding and testing have been developed and recorded in the form. Since the proposed change aims to allow more flexibility in the PXR system, its implementation priority has been ranked as *important*, and it will be developed for the next system release.

Step 5. Document the maintenance process itself.

Once evaluation of the approved changes has been carried out, their specification is started and documented using the Maintenance Specification form. Figure 8.5 shows the form *ms-1*, which has been developed to specify the approved changes *ca_cp-1* and *ca_cp-2*. The field *Specification of change* contains the specification of the enhanced system operations. Additionally, the fields *Integration tests* and *System tests* specify the tests which have to be carried out in order to complete the system release.

Step 6. Document the software components to be modified.

During evaluation of the approved changes, the software components that need to be modified have been detected and linked to the Change Approval forms. Since the software component *arguments.p* is involved in the proposed change, it has to be documented, adding to the system documentation. The corresponding Module Design form for *arguments.p* is called *arguments-design*. Therefore, the Module Design form *arguments-design*, and Module Source Code form *arguments.p*, have to

```
------------------------------------------------------------------
MS identification: ms-1    date: 28/01/92
MS formulated by:  des3mmc
------------------------------------------------------------------
MS status: in_development     status date: 28/01/92
------------------------------------------------------------------
1. Related CAs:
         ca_cp-1
         ca_cp-2
2. Specification of change: Procedure Usage: change write command to
   writeln('usage: pxr [ [-l] [-{t|i|f}{a|s}] ] filename');.
   Procedure Three_char_param accepts f option by scanning for 'f'.
   Change procedure GetArguments to set up defaults and check
   for '-' before options.
3. Affected sw components id.:
         arguments-design
4. Consequences of change: None
5. Other necessary changes:
6. Integration tests: commands:'pxr -fs test.p' and 'pxr -fa test.p'
7. System tests: commands:'pxr' and 'pxr test.p'
------------------------------------------------------------------
```

Figure 8.5: A Maintenance Specification form

be fully developed so as to complete the system release. Figure 8.6 displays the Module Design form *arguments-design*.

```
------------------------------------------------------------------
MD identification: arguments-design     date: 28/01/92
MD designed by:  des3mmc
------------------------------------------------------------------
MD status: frozen      status date: 28/01/92
MD baseline established by: des3mmc
------------------------------------------------------------------
1. Module purpose: Get and parse parameters from command line
2. Algorithms outline: None
3. Interface definitions: Uses Unix argc and argv through
   the nonstan interface
4. Test plans: None
------------------------------------------------------------------
```

Figure 8.6: A Module Design form

Step 7. Implement and test the changes on the software components.

The software component *arguments.p* has been modified using RCS and *make* in order to generate the first release of the software system. The planned tests in the Module Design and Maintenance Specification forms have been performed and recorded in the Module Source Code form and Configuration Release form.

Step 8. Initial evaluation of forms.

The initial evaluation of the auditing process is then carried out by effectuating the forms of the system release. The initial evaluation ensures the completeness of the forms. In addition, the tests which have to be performed are checked. The *checker* program is run to ensure that all the modified software components have been updated in the COMFORM object base. In order to illustrate the version control applied to the forms, Figure 8.7 shows the Module Source Code form *arguments.p*, along with its alternative. The original *arguments.p* form has been frozen and contains one version and one alternative, as displayed in the *status* section. The alternative has been completed and effectuated. The *Description of modification* field of the first version of the Module Source Code form (*arguments.p-1.1*) is *Initial revision* as automatically created by the RCS tool. In the *arguments.p-1.2*, the additional field *Description of modification* describes that it has been created in response to the changes required by the change proposals *cp-1* and *cp-2*.

Step 9. Establish the baselines of forms.

Establishment of the baselines is carried out assuring the consistency and traceability of the configuration to be released. At this stage, all the forms of the system release have already been effectuated. The Module Source Code form *arguments.p* is the first to have its baseline established, followed by its corresponding Module Design form, *arguments-design*. The Change Approval forms *ca_cp-1* and *ca_cp-2*, followed by the Maintenance Specification form *ms-1* have their baselines established subsequently.

```
---------------------------------------------------------------------
SC identification: arguments.p     date: 28/01/92
SC implemented by: des3mmc
---------------------------------------------------------------------
SC status: frozen      status date: 28/01/92
Alternatives/versions:
           arguments.p-1.1  (frozen)
           arguments.p-1.2  (effective)
SC baseline established by: des3mmc
---------------------------------------------------------------------
1. Corresponding MD:
         arguments-design
2. Tests outcome: OK
3. Comments: This should be independent of OS (must be like
   Unix?!?)
4. SC understood by: mm
---------------------------------------------------------------------



---------------------------------------------------------------------
SC version identification: arguments.p-1.2     date: 28/01/92
SC version author: des3mmc
---------------------------------------------------------------------
SC version status: effective     status date: 28/01/92
---------------------------------------------------------------------
0. Description of modification: Added code to deal with f
   option and to let the program have default options.
1. Corresponding MD:
         arguments-design
2. Tests outcome: OK
3. Comments:  None
4. SC understood by: mm
---------------------------------------------------------------------
```

Figure 8.7: A Module Source Code form and an alternative

Step 10. Release the system configuration to the users.

At last the Configuration Release baseline has been established and released to the users. Figure 8.8 shows the Configuration Release form with its fields completed,

```
-----------------------------------------------------------------
CR identification: pxr-1.1     date: 28/01/92
CR created by:  des3mmc
-----------------------------------------------------------------
CR status: frozen      status date: 28/01/92
CR baseline established by: des3mmc
-----------------------------------------------------------------
          Comprises:
CPs:
     cp-1
     cp-2
CAs:
     ca_cp-1
     ca_cp-2
MSs:
     ms-1
Is composed of:
         arguments-design
         arguments.p-1.2
1. Integration tests outcome: Ok
2. System tests outcome: Ok
3. Configuration distributed to: Andrew, Thomas and Joan
-----------------------------------------------------------------
```

Figure 8.8: A Configuration Release form

enabling the first system release be available to the users. When the system release reaches this stage, COMFORM ensures that the maintenance process has been documented and controlled. Furthermore, the involved software components have had their documentation improved. The auditing process ensures that the necessary tests have been performed, and that the completeness, consistency and traceability checks in the forms have been accomplished.

Step 11. Generate reports.

The report generator tool can be run at any time to obtain reports with specific information from the COMFORM object base. Figure 8.9 shows a report displaying useful information to the project manager. This report displays the Module Design forms created by a particular author. It shows that the author *des3mmc* has created two Module Design forms, *arguments-design* and *paramlist-design*, to improve the documentation of the PXR system.

```
                      System: PXR
                      -----------

                      Author: des3mmc


Module Design
-----------------
arguments-design
paramlist-design

Total Module Design = 2
```

Figure 8.9: A managerial report

The parameters chosen to obtain this report are shown in Table 8.1. According to the chosen parameters, the name of the current software system and the total of forms presented in the report should be displayed (parameters 1 and 2). In addition, the only information to be displayed about the chosen type of form, that is, Module Design (parameter 4.2), is the form name (parameter 3.1). Nevertheless, not all the Module Design forms should be displayed, but only those created by a particular author (parameter 5.3).

1. Name of current cluster				display (Y/N)	Y
2. Total of forms				display (Y/N)	Y
3. Information to be displayed					
3.1 Form name	x	3.2 Creation date		3.3 Author of creation	
3.4 Current status		3.5 Status date		3.6 Author of baseline	
4. Type of forms to be displayed					
4.1 Configuration Release		4.2 Module Design	x	4.3 Module Source Code	
4.4 Maintenance Specification		4.5 Change Approval		4.6 Change Proposal	
5. Optional parameters to be matched					
5.1 Form name		5.2 Creation date		5.3 Author of creation	x
5.4 Current status		5.5 Status date		5.6 Author of baseline	

Table 8.1: Parameters of a managerial report

1. Name of current cluster				display (Y/N)	Y
2. Total of forms				display (Y/N)	N
3. Information to be displayed					
3.1 Form name	x	3.2 Creation date	x	3.3 Author of creation	x
3.4 Current status	x	3.5 Status date	x	3.6 Author of baseline	
4. Type of forms to be displayed					
4.1 Configuration Release		4.2 Module Design		4.3 Module Source Code	
4.4 Maintenance Specification	x	4.5 Change Approval	x	4.6 Change Proposal	x
5. Optional parameters to be matched					
5.1 Form name		5.2 Creation date		5.3 Author of creation	
5.4 Current status		5.5 Status date		5.6 Author of baseline	

Table 8.2: Parameters to report the forms of a Configuration Release

Figure 8.10 shows a report, with information for the maintainer, containing the forms which have already been created in a particular configuration release. In this report, it shows that two Change Proposal forms *cp-1* and *cp-2*, associated with the Configuration Release form *pxr-1.1*, have been approved (*frozen* status) and have generated two Change Approval forms *ca_cp-1* and *ca_cp-2*, which are *in development.* In addition, *ms-1* is the Maintenance Specification form which contains the specification of the proposed changes.

The parameters to create this report are displayed in Table 8.2. In this report, the

```
                 Configuration Release: pxr-1.1
                 ------------------------------


Change Proposal      Creation Date Author  Status         Status Date
---------------------------------------------------------------------
cp-1                  28/01/92      des3mmc frozen          28/01/92
cp-2                  28/01/92      des3mmc frozen          28/01/92


Change Approval      Creation Date Author  Status         Status Date
---------------------------------------------------------------------
ca_cp-1               28/01/92      des3mmc in_development 28/01/92
ca_cp-2               28/01/92      des3mmc in_development 28/01/92


Maintenance          Creation Date Author  Status         Status Date
Specification
---------------------------------------------------------------------
ms-1                  28/01/92      des3mmc in_development 28/01/92
```

Figure 8.10: The forms of a Configuration Release

name of the chosen configuration release is displayed (parameter 1), but the total of forms has not been required (parameter 2). The forms of a particular configuration release may be Change Proposal, Change Approval or Maintenance Specification forms, represented by the parameters 4.6, 4.5 and 4.4 respectively. In addition, the information to be displayed about the chosen types of forms consists of the form name, creation date, author of creation, current status, and status date, as represented by the parameters 3.1, 3.2, 3.3, 3.4 and 3.5 respectively.

Since COMFORM allows the incremental redocumentation of the software system, a useful report to be obtained is the one displaying the software components which have been modified and documented. Figure 8.11 shows such a report, displaying all the Module Design and Module Source Code forms of the PXR system. In this report, all the software components of the system are represented by a Module Source Code form. Additionally, as the *arguments.p* and *paramlist.p* are being modified, their corresponding Module Design forms *arguments-design* and *paramlist-design* have been generated in order to improve their documentation. Furthermore, the status of these forms evolves from *in development* to *effective* and *frozen*, as they are documented.

The current implementation of the report generator tool has one restriction; that is, it only allows the display of information of a single cluster of objects at a time. Therefore, larger reports cannot be obtained directly. One way to overcome such a limitation is to combine one or more reports, so that the final one is larger and contains more information. Consequently, all the forms associated with a particular software system can only be obtained by grouping together reports, so that the first report displays the more generic information associated with the system; for instance, Configuration Release, Module Design and Module Source Code forms.

System: PXR

Module Design	Creation Date	Author	Status	Status Date
arguments-design	28/01/92	des3mmc	effective	28/01/92
paramlist-design	28/01/92	des3mmc	in_development	28/01/92

Module Source Code	Creation Date	Author	Status	Status Date
arguments.p	28/01/92	des3mmc	effective	28/01/92
error.p	28/01/92	des3mmc	in_development	28/01/92
expression.p	28/01/92	des3mmc	in_development	28/01/92
getconstant.p	28/01/92	des3mmc	in_development	28/01/92
gettoken.i	28/01/92	des3mmc	in_development	28/01/92
gettoken.p	28/01/92	des3mmc	in_development	28/01/92
gettype.p	28/01/92	des3mmc	in_development	28/01/92
idstack.p	28/01/92	des3mmc	in_development	28/01/92
includestack.p	28/01/92	des3mmc	in_development	28/01/92
linenumber.p	28/01/92	des3mmc	in_development	28/01/92
literal.p	28/01/92	des3mmc	in_development	28/01/92
paramlist.p	28/01/92	des3mmc	in_development	28/01/92
print.p	28/01/92	des3mmc	in_development	28/01/92
pxr.p	28/01/92	des3mmc	in_development	28/01/92
readblock.p	28/01/92	des3mmc	in_development	28/01/92
readconst.p	28/01/92	des3mmc	in_development	28/01/92
readfile.i	28/01/92	des3mmc	in_development	28/01/92
readfile.p	28/01/92	des3mmc	in_development	28/01/92
readlabel.p	28/01/92	des3mmc	in_development	28/01/92
readprogram.p	28/01/92	des3mmc	in_development	28/01/92
readroutine.p	28/01/92	des3mmc	in_development	28/01/92
readtype.p	28/01/92	des3mmc	in_development	28/01/92
readvar.p	28/01/92	des3mmc	in_development	28/01/92
statement.p	28/01/92	des3mmc	in_development	28/01/92
stripzeroes.p	28/01/92	des3mmc	in_development	28/01/92
symbol.i	28/01/92	des3mmc	in_development	28/01/92
symbol.p	28/01/92	des3mmc	in_development	28/01/92
symbolstack.p	28/01/92	des3mmc	in_development	28/01/92
variable.p	28/01/92	des3mmc	in_development	28/01/92

Figure 8.11: The Module Design and Module Source Code forms of the PXR system

The following reports would then display the information contained in each of the System Release cluster of objects of the software system, so that forms related to each Configuration Release can be obtained. Figure 8.12 shows a final report obtained from the combination of three reports. The first one contains the Configuration Release forms of the PXR system (*pxr-1.1* and *pxr-1.2*), whereas the second and third reports contain information about the forms associated with these two Configuration Release forms.

8.3 Comments on the COMFORM prototype

The COMFORM prototype has been developed in order to provide automated support necessary to evaluate the underlying method. Since it is a prototype tool, it contains some shortcomings as described below.

Time constraints made it impossible to develop an environment which shared the features of generalized CASE technology; for instance, the user friendly interface, used to manipulate and browse through the documentation produced.

The user interface of the COMFORM prototype is not graphical, because it has not been the objective of this thesis to develop the prototype using windows and pop-up menus. An improvement which should be made to the prototype is that the user interface could be made more user-friendly and driven intuitively by mouse, icons and menus, rather than being driven by an old-fashioned, text-based command structure.

The current implementation of the report generator tool does not allow the user

```
                        System: PXR
                        -----------

Configuration Release  Creation Date  Author  Status          Status Date
--------------------------------------------------------------------------
pxr-1.1                 28/01/92      des3mmc in_development 28/01/92
pxr-1.2                 28/01/92      des3mmc in_development 28/01/92

                        Configuration Release: pxr-1.1
                        ------------------------------

Change Proposal Creation Date Author Status  Status Date Author Baseline
--------------------------------------------------------------------------
cp-1              28/01/92   des3mmc frozen           28/01/92  des3mmc
cp-2              28/01/92   des3mmc frozen           28/01/92  des3mmc

Change Approval Creation Date Author Status  Status Date Author Baseline
--------------------------------------------------------------------------
ca_cp-1           28/01/92   des3mmc in_development 28/01/92
ca_cp-2           28/01/92   des3mmc in_development 28/01/92

Maintenance     Creation Date Author Status  Status Date Author Baseline
Specification
--------------------------------------------------------------------------
ms-1              28/01/92   des3mmc in_development 28/01/92

                        Configuration Release: pxr-1.2
                        ------------------------------

Change Proposal Creation Date Author Status  Status Date Author Baseline
--------------------------------------------------------------------------
cp-3              28/01/92   des3mmc frozen           28/01/92  des3mmc
cp-4              28/01/92   des3mmc in_development 28/01/92
cp-5              28/01/92   des3mmc in_development 28/01/92

Change Approval Creation Date Author Status  Status Date Author Baseline
--------------------------------------------------------------------------
ca_cp-3           28/01/92   des3mmc in_development 28/01/92

Maintenance     Creation Date Author Status  Status Date Author Baseline
Specification
--------------------------------------------------------------------------
ms-2              28/01/92   des3mmc in_development 28/01/92
```

Figure 8.12: The maintenance history of the PXR system

to generate reports containing all the information in the COMFORM object base. Information can only be obtained by choosing the parameters available in the report generator tool. Moreover, the presentation of the contents of the reports could be improved by sorting out the information according to the users requests.

The version control implemented in the COMFORM prototype is such that versions are always generated from the initial form. The concepts of revisions and variations, discussed in Chapter 5, have not been implemented. Such concepts are of great usage to Module Design and Module Source Code forms.

Additional features could be incorporated into the COMFORM prototype in order to produce a more automated system. As far as the validation process is concerned, tests which should be carried out during the maintenance process have no automated support, so that the prototype relies on the maintainers input in order to ensure that the modifications have gone through the testing process correctly.

The change approval process also lacks automated help; the resources spent on system changes need to be estimated. Again, COMFORM has to rely on the maintainer's manual help to fill in these fields. An automated tool could also be incorporated into the prototype, so that a more reliable response could be obtained.

Although a considerable amount of work remains to be done, the development of a prototype capable of showing how COMFORM could support a software maintenance environment has been implemented. In view of the aim of COMFORM is to generate a method applying the SCM discipline, its automation should consist of a framework representing the support for the management of tasks required by the maintenance process. Consequently, features which could be implemented, as some have been suggested in this section, are not part of the method, since SCM aims to

manage and control the maintenance activities and not perform them; but the automation of such features would certainly improve the reliability of the automation of COMFORM.

From this brief experience of employing COMFORM, it can be concluded that it contains some characteristics which might be considered inexpedient by some. However, it is believed that the advantages of using COMFORM outweigh the drawbacks, as can be seen in the next chapter.

Chapter 9

Conclusions

This chapter summarizes the research which has been carried out in this thesis, and suggests possible directions and ideas for future research related to this work.

9.1 Review of Work

With an ever increasing body of software to maintain, the aim of this thesis was the development of a method for maintaining existing software systems and which would generally improve the software maintenance process. To develop ideas for this method, several areas were surveyed. The initial chapters of this thesis covered the required background to the introduction and development of COMFORM. These chapters described: software maintenance concepts and techniques, the SCM

discipline, existing software maintenance models and modelling techniques for the formalization of the proposed method.

The goal of COMFORM is to provide a consistent and structured approach to software maintenance. It aims to exert control over an existing software system, whilst simultaneously incrementally redocumenting it. Accordingly, the major achievements of this research are:

- The development of a new method - COMFORM - which provides guidelines and procedures for the software maintenance process, by applying the SCM discipline (Chapter 5).
- The formalization of COMFORM by the use of ORM. A benefit of this formalization is the provision of easy visualization of information representation. In addition, the cluster mechanism enabled the representation of different levels of abstraction, which eventually facilitated the method implementation (Chapter 6).
- The implementation of a prototype, which forms the foundation of a software maintenance environment, consisting of a framework which provides automated support for the concepts and capabilities of COMFORM (Chapter 7).

In order to develop the method, the maintenance process was formally structured into a sequence of phases, with an auditing process at the end of each phase. SMM institutes this structural model of COMFORM, and is the backbone of this work. By determining the SMM phases, the steps which a change should follow during maintenance were clearly defined. The forms, which are the products of each phase, represent maintenance history and an abstraction of the operational product neces-

sary to improve the documentation of a poorly-documented existing software system. The notion of incremental redocumentation was achieved in the method, as the software components, which undergo modifications, need to have their corresponding Module Design and Module Source Code forms documented in order to complete a system release.

9.2 The Assessment of COMFORM

The idea of using forms associated with each SMM phase provides a number of benefits, as detailed below:

- This approach strengthens the application of the SCM discipline, as the forms represent the products of each SMM phase. As a result, the performance of the SCM functions in pre-defined documents is facilitated.
- The following of SMM phases institutes a change control procedure to monitor changes.
- The completeness checks are assured by the use of forms, since no essential details are omitted.
- The consistency checks are also ensured by the use of forms, as the information required by forms are provided by other forms in the system.
- The traceability between phases is facilitated by establishing the relationships between components of different phases in the forms.

- The uniformity of information is guaranteed, as forms are pre-defined, avoiding inconsistency and unnecessary differences.

The SCM discipline was central to the development of COMFORM, because in essence, this discipline is concerned with change; in particular, how to control change, how to manage software systems subject to change and how to release these changed software systems to users. To address these issues, the four SCM functions (identification, control, status accounting and auditing) were considered and integrated into COMFORM.

During the software configuration identification function, SCIs are identified, with links and dependencies between them established by the method. The new forms generated during COMFORM use are identified as SCIs. Additionally, new software components (or versions of existing ones) of an existing software system are identified as SCIs during their introduction as Module Source Code forms to COMFORM.

The software configuration control function is accomplished by following the SMM phases, which starts with a change proposal and ends with a system release. The procedures controlled by this function for proposing, evaluating, reviewing, approving and implementing changes to a baseline, are performed in order to fill in the required forms to generate a system release. The automation of the method enforced the correct realization of this function.

The software configuration status accounting function is accomplished by storing the information in a reliable way, by filling in SMM forms. The retrieval of system information is obtained by a number of reports, which aid project managers and maintainers.

The software configuration auditing function within COMFORM is the process which determines the overall acceptability of the proposed baseline at the end of each SMM phase. It aims to establish the baselines of SMM forms, and entails the performance of checks such as completeness, correctness, consistency and traceability on the forms. Without the establishment of the baseline of all forms of a system release (which ends with the establishment of the baseline of the Configuration Release form), the modifications cannot be released to the users.

The formalization of COMFORM proved to be an effective strategy for the development of the method. The representation of SMM forms as object types outlined another perspective of the software maintenance process. Moreover, the use of the cluster mechanism provided a further aid, in that SMM forms could be represented at different levels of abstraction. The use of the cluster mechanism also enabled the performance of version control on the forms. Additionally, in order to apply the scheduled release notion, change proposals were grouped in system releases so that they have to be more carefully investigated before being introduced to COMFORM. As a result, the Change Proposal, Change Approval and Maintenance Specification forms were grouped in such a way that they express the history of each system release; while the Module Design and Module Source Code forms represent the documentation of the software components, making up the incremental documentation of the method.

In Chapter 3, the concept of software process modelling, along with the desirable characteristics of a suitable modelling methodology, were described. In view of these characteristics of the software development process, the capabilities of COMFORM were further analysed, in order to evaluate the approach taken by the proposed method. A large number of COMFORM characteristics emphasized the coverage

that it gives to the software maintenance process, as shown below.

- The maintenance process of COMFORM supports a number of perspectives. Functional perspective is obtained through SMM phases, which represent the activities being performed, along with their corresponding forms containing information pertaining to each involved task. Conceptual data modelling perspective is achieved by the formalization of information using ORM. These perspectives employ diagrams for information representation. Additionally, they support different levels of abstraction. SMM phases alone represent the different levels of abstraction. These phases in the formalization are represented by the cluster hierarchy reflecting the levels of abstraction.

- COMFORM makes use of pre-defined forms for system documentation, enabling compendious descriptions. With the use of forms, quality assurance checks such as consistency, completeness, traceability and correctness are easily obtained and automated. The forms also provide the means for defining the syntax of documentation. Moreover, form fields semantically represent the objects, attributes and relationships of the conceptual data model. Therefore, the forms, as proposed in this work, are the constructs which ease the method's automation.

- The COMFORM prototype provides the basic automated support for the method, affording the capabilities of automatically recording the data about the maintenance process through the manipulation of forms.

9.3 Directions for Further Research

Following on from the work reported in this thesis, there is scope for additional research in a number of directions which could be usefully undertaken.

In a very recent reference, Edelstein and Mamone [36] report on the development of a standard for software maintenance by the IEEE P1219 Working Group. This standard aims to provide a framework, containing precise terminology and processes for the consistent application of technology (tools, techniques and methods) in order to solve software maintenance problems. This standard comprises three sections, in which the first section delineates the steps involved in software maintenance, including the input and output required for the steps. The other two sections are appendices to the first one, and consist of maintenance guidelines and supporting maintenance technology.

It is interesting to note that in one way or another, COMFORM tackles most of the capabilities defined by the software maintenance process model suggested by this standard. In line with the proposed standard, however, COMFORM seems to need improvement in the area of testing, as it only requires that integration and system testings be developed and/or modified during the specification of changes, and a system release only being made available when these tests have been performed.

In order to emphasize testing in COMFORM, this could be achieved by an additional phase which would be a pre-requisite of the *system release* phase. The input of this phase could be the Maintenance Specification form and the forms associated with the changed software components. The output would be a report confirming that:

- The software system executes correctly.
- All software components affected by the change have been tested.
- The implemented changes correspond to those in the Maintenance Specification form.
- Software components, unrelated to the changes, have not been altered.
- The existing standards have been adhered to.

Another interesting area for further research would be that of COMFORM supporting the behavioural perspective of the software maintenance process. This perspective would describe *when* the activities of COMFORM should be accomplished. The formalism for this perspective would represent aspects such as iteration among SMM phases, complex decision-making conditions, entry criteria and exit criteria for the phases. It could also represent and convey parallelism and address the concept of synchronization. For example, integration testing could only begin when all software components have passed unit testing. The development of the behavioural perspective in COMFORM would enable the simulation of process behaviour directly from the representation.

An important question is how the described method suits every type of existing software system which needs to be maintained? Form fields in COMFORM have been created in such a way as to attempt covering all maintenance needs. However, these fields may differ dramatically from one type of software system to another. It would be better if more flexible implementation could be carried out, so that customizable fields could be easily incorporated into COMFORM. In order to achieve this flexibility, the COMFORM data model should be implemented in a meta-system

approach, where the fields of forms (attribute types) could be easily changed and added to suit the particular software systems being maintained.

Finally, new experiments with COMFORM will provide additional results in order to fully evaluate its capabilities, which can only be truly tested when applied to the task of maintaining substantial existing software systems.

While this thesis has not had the last word as far as software maintenance research goes in tackling the maintenance problem, it has made a positive contribution to this field; other related topics could also be put forward to further maintenance research.

Bibliography

[1] Alderson, A., Bott, M. F. and Falla, M. E., **"The Eclipse Object Management System"**, *Software Engineering Journal, 1 (1)*, Jan. 1986, pp. 39-42

[2] Ambriola, V., Bendix, L. and Ciancarini, P., **"The Evolution of Configuration Management and Version Control"**, *Software Engineering Journal*, Nov. 1990, pp. 303-310

[3] Ambriola, V. and Bendix, L. , **"Object-Oriented Configuration Control"**, *Software Engineering Notes, 14 (7)*, Nov. 1989, pp. 133-136

[4] Arthur, L. J., **Software Evolution - The Software Maintenance Challenge**, *John Wiley & Sons*, 1988

[5] Babich, W. A., **Software Configuration Management - Coordination for Team Productivity**, *Addison-Wesley*, 1986

[6] Belkatir, N., Melo, W. L., Estublier, J. and Nacer, M. A., **"Supporting Software Maintenance Evolution Process in the Adele System"**, *Proc. of the 30th Annual ACM Southeast Conference, Raleigh, NC*, Apr. 8-10, 1992

[7] Belkatir, N. and Estublier, J., **"Experience with a Data Base of Programs"**, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Dec. 1986, pp. 84-91

[8] Bennett, K. H., **"Automated Support of Software Maintenance"**, *Information and Software Technology, 33 (1)*, Jan./Feb. 1991, pp. 74-85

[9] Bennett, K. H., Cornelius, B., Munro, M. and Robson, D., **"Software Maintenance"**, *Software Engineer's Reference Book, edited by J. A. McDermid, Butterworth-Heinemann*, 1991, Section 20

[10] Bennett, K. H., **"The Software Maintenance of Large Software Systems: Management, Methods and Tools"**, *Software Engineering for Large Software Systems, edited by B. A. Kitchenham, Elsevier Science Publishers Ltd*, 1990, pp. 1-26

[11] Bersoff, E. H. and Davis, A. M., **"Impacts of Life Cycle Models on Software Configuration Management"**, *Communications of the ACM, 34 (8)*, Aug. 1991, pp. 104-118

[12] Bersoff, E. H., **"Elements of Software Configuration Management"**, *IEEE Transactions on Software Engineering, 10 (1)*, 1984, pp. 79-87

[13] Bersoff, E. H., Henderson, V. D. and Siegel, S. G., **Software Configuration Management - An Investment in Product Integrity**, *Prentice-Hall*, 1980

[14] Biggerstaff, T. J., **"Design Recovery for Maintenance and Reuse"**, *Computer*, Jul. 1989, pp. 36-49

[15] Blum, B. I., **"Documentation for Maintenance: A Hypertext Design"**, *Proc. of Conference on Software Maintenance-1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 23-31

[16] Boehm, B. W., **"Software Engineering"**, *IEEE Trans. Computer, 25 (12)*, Dec. 1976, pp. 1226-1241

[17] Booch, G., **Object-Oriented Design with Applications**, *The Benjamin/Cummings Publishing Company, Inc.*, 1991

[18] Bott, F. (Ed.), **Eclipse - An Integrated Project Support Environment**, *Peter Peregrinus Ltd.*, 1989

[19] Broadey K. M., Colbrook, A., Munro, M. and Robson, D. J., **"Block-structured Cross-referencers for Pascal and C"**, *University Computing, 11 (3)*, Sep. 1989, pp. 121-128

[20] Brown, A., **Object-Oriented Databases - Application in Software Engineering**, *The McGraw-Hill Book Company (UK) Limited*, 1991

[21] Bryan, W. L., Siegel, S. G. and Whiteleather, G. L., **"Auditing Throughout the Software Life Cycle: A Primer"**, *Computer*, Mar. 1982, pp. 57-67

[22] Buckle, J. K., **Software Configuration Management**, *Macmillan Education*, 1982

[23] Capretz, M. A. M. and Munro, M., **"Software Configuration Management Issues in the Maintenance of Existing Systems"**, *Journal of Software Maintenance: Research and Practice*, 1992, Accepted for Publication

[24] Capretz, M. A. M. and Munro, M., **"COMFORM - A Software Maintenance Method Based on the Software Configuration Management Discipline"**, *Proc. of Conference on Software Maintenance-1992, Orlando, Florida*, Nov. 9-12, 1992, Accepted for the Conference

[25] Capretz, M. A. M., **Automation of Software Configuration Management**, *MSc thesis, FEC-UNICAMP, Campinas, Sao Paulo, Brazil, (in Portuguese)*, Aug. 1988

[26] Chapin, N., **"Software Maintenance: A Different View"**, *AFIPS Conf. Proc. 54 National Computer Conference*, 1985, pp. 509-513

[27] Chen, Y-F. and Ramamoorthy, C. V., **"The C Information Abstractor"**, *Proc. of IEEE COMPSAC*, 1986, pp. 291-298

[28] Chen, P. P-S., **"The Entity-Relationship Model - Toward a Unified View of Data"**, *ACM Transactions on Database Systems, 1 (1)*, Mar. 1976, pp. 9-36

[29] Chikofsky, E. J. and Cross II, J. H., **"Reverse Engineering and Design Recovery: A Taxonomy"**, *IEEE Software, 7 (1)*, Jan. 1990, pp. 13-17

[30] Choi, S. C. and Scacchi, W., **"SOFTMAN: Environment for Forward and Reverse CASE"**, *Information and Software Technology, 33 (9)*, Nov. 1991, pp. 664-674

[31] Choi, S. C. and Scacchi, W., **"Assuring the Correctness of Configured Software Descriptions"**, *Proc. of the 2nd. International Workshop on Software Configuration Management, Princeton, New Jersey*, Oct. 24, 1989, pp. 66-75

[32] Codd, E. F., **"A Relational Model of Data for Large Shared Data Banks"**, *Communications of the ACM, 13 (6)*, Jun. 1970, pp. 377-387

[33] Collofello, J. S. and Orn, M., **"A Practical Software Maintenance Environment"**, *Proc. of Conference on Software Maintenance-1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 45-51

[34] Collofello, J. S. and Buck, J. J.., **"Software Quality Assurance for Maintenance"**, *IEEE Software*, Sep. 1987, pp. 46-51

[35] Desclaux, C. and Ribault, M., **"MACS: Maintenance Assistance Capability for Software A K.A.D.M.E."**, *Proc. of Conference on Software Maintenance-1991, Sorrento, Italy*, Oct. 15-17, 1991, pp. 2-12

[36] Edelstein, D. V. and Mamone, S., **"A Standard for Software Maintenance"**, *Computer*, Jun. 1992, pp. 82-83

[37] Estublier, J., **"A Configuration Manager: The Adele Data Base of Programs"**, *Proc. of Workshop on Software Engineering Environments for Programming-in-the-large*, Jun. 1985, pp. 140-147

[38] Estublier, J., Ghoul, S. and Krakowiak, S., **"Preliminary Experience with a Configuration Control System for Modular Programs"**, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Apr. 1984, pp. 149-156

[39] Feldman, S. I., **"Make - A Program for Maintaining Computer Programs"**, *Software Practice and Experience, 9*, 1979, pp. 255-265

[40] Fletton, N. T., **Documentation for Software Maintenance and the Redocumentation of Existing Systems**, *MSc thesis, SEAS (Computer Science), University of Durham, Durham, England*, Sep. 1988

[41] Fletton, N. T. and Munro, M., **"Redocumenting Software Systems using Hypertext Technology"**, *Proc. of Conference on Software Maintenance-1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 54-59

[42] Foster, J. R., Jolly, A. E. P. and Norris, M. T., **"An Overview of Software Maintenance"**, *British Telecom Technology Journal, 7 (4)*, Oct. 1989, pp. 37-46

[43] Foster, J. R. and Munro, M., **"A Documentation Method Based on Cross-Referencing"**, *Proc. of Conference on Software Maintenance-1987, Austin, Texas*, Sep. 21-24, 1987, pp. 181-185

[44] Gadd, R. J., **"ReForm - From Assembler to Z Using Formal Transformations"**, *Fourth Software Maintenance Workshop, Durham*, Sep. 1990

[45] Gallagher, K. B., **"Using Program Slicing in Software Maintenance"**, *PhD Thesis, University of Maryland, Faculty of the Graduate School (TR CS-90-05)*, Jan. 1990

[46] Gane, C. and Sarson, T., **Structured System Analysis: Tools and Techniques**, *Prentice Hall, Englewood Cliffs, New Jersey*, 1979

[47] Garg, P. K. and Scacchi, W., **"A Hypertext System to Manage Software Life-Cycle Documents"**, *IEEE Software, 7 (3)*, May 1990, pp. 90-98

[48] Garg, P. K. and Scacchi, W., **"Ishys Designing an Intelligent Software Hypertext System"**, *IEEE Expert, 4 (3)*, Fall 1989, pp. 52-63

[49] Garg, P. K. and Scacchi, W., **"A Software Hypertext Environment for Configured Software Descriptions"**, *Proc. of International Workshop on Software Version and Configuration Control, Grassau, W. Germany*, Jan. 1988, pp. 326-343

[50] Garland, J. K. and Calliss, F. W., **"Improved Change Tracking for Software Maintenance"**, *Proc. of Conference on Software Maintenance-1991, Sorrento, Italy*, Oct. 15-17, 1991, pp. 32-41

[51] Habermann, A. N. and Notkin, D., **"Gandalf Software Development Environments"**, *IEEE Trans. on Software Engineering, 12 (12)*, Dec. 1986, pp. 1117-1127

[52] Hager, J., **"Developing Maintainable Systems: A Full Life Cycle Approach"**, *Proc. of Conference on Software Maintenance-1989, Miami, Florida*, October 16-19, 1989, pp. 271-278

[53] Harrison, W. and Magel, K. I., **"A Complexity Measure Based on Nesting Level"**, *ACM SIGPLAN Notices*, 1981, pp. 63-74

[54] Hinley, D. S. and Bennett, K. H., **"Using a Model to Manage the Software Maintenance Process"**, *Proc. of Conference on Software Maintenance-1992, Orlando, Florida*, Nov. 9-12, 1992, Accepted for the Conference

[55] Horowitz, E. and Williamson, R. C., **"SODOS: A Software Documentation Support Environment - Its Definition"**, *IEEE Trans. on Software Engineering, 12 (8)*, Aug. 1986, pp. 849-859

[56] Horowitz, E. and Williamson, R. C., **"SODOS: A Software Documentation Support Environment - Its Use"**, *IEEE Trans. on Software Engineering, 12 (11)*, Nov. 1986, pp. 1076-1087

[57] ANSI/IEEE Standard 729, **IEEE Standard Glossary of Software Engineering Terminology**, *IEEE*, 1983

[58] Held at Moretonhampstead, Devon, UK, **Proc. of the 4th International Software Process Workshop: Representing and Enacting the Software Process, 1988**, *published as ACM Software Engineering Notes, 14 (4)*, Jun. 1989

[59] Kaiser, G. E. and Habermann, A. N., **"An Environment for System Version Control"**, *Compcon'83 IEEE Computer Society, San Francisco,* Feb. 1983, pp. 415-420

[60] Katz, R. H. and Lehman, T. J., **"Database Support for Versions and Alternatives of Large Design Files"**, *IEEE Transactions on Software Engineering, 10 (2),* Mar. 1984, pp. 191-200

[61] Kellner, M. I., **"Multiple-Paradigm Approaches for Software Process Modeling"**, *Proc. of the 7th International Software Process Workshop: Communication and Coordination in the Software Process,* Oct. 15-18, 1991

[62] Kellner, M. I. and Hansen, G. A., **"Software Process Modeling: A Case Study"**, *Proc. of the Twenty-Second Annual Hawaii International Conference on System Sciences, Vol. II - Software Track, edited by Shriver, B. D., IEEE,* Jan 3-6, 1989, pp. 175-188

[63] Kenning, R. J. and Munro, M., **"Understanding the Configurations of Operational Systems"**, *Proc. of Conference of Software Maintenance-1990, San Diego, California,* Nov. 26-29, 1990, pp. 20-27

[64] Kenning, R. J. and Munro, M., **"PISCES - An Inverse Configuration Management System"**, *Software Reuse and Reverse Engineering in Practice, edited by D. A. V. Hall, Chapman & Hall,* 1992, pp. 489-501

[65] Kernighan, B. W. and Ritchie, D. M., **The C Programming Language**, *Prentice-Hall, Inc.*, 1978

[66] Khorshid, W. and Rajlich, V., **"VIPEG: A Generator of Environments for Software Maintenance"**, *Proc. of 14th Annual International Computer*

Software and Applications Conference (COMPSAC) 1990, Chicago, USA, 1990, pp. 471-476

[67] Khoshafian, S. and Abnous, R., **Object-Orientation - Concepts, Languages, Databases, and User Interfaces**, *John Wiley & Sons, Inc.*, 1990

[68] King R., **"My Cat is Object-Oriented"**, *Object-Oriented Concepts, Databases, and Applications, ed. by W. Kim and F. H. Lochovsky, Addison-Wesley Publishing Company*, 1989, pp. 23-30

[69] Lampson, B. W. and Schmidt, E. E., **"Organizing Software in a Distributed Environment"**, *SIGPLAN Notices, ACM, 18 (6)*, Jun. 1983, pp. 1-13

[70] Lamsweerde, A., Buyse, M., Delcourt, B., Delor, E., Ervier, M., Schayes, M. C., Bouquelle, J. P., Champagne, R., Nisole, P. and Seldeslachts, J., **"The Kernel of a Generic Software Development Environment"**, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Dec. 1986, pp. 208-217

[71] Landis, L. D., Hyland, P. M., Gilbert, A. L. and Fine, A. J., **"Documentation in a Software Maintenance Environment"**, *Proc. of Conference on Software Maintenance - 1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 66-73

[72] Lano, K. and Haughton, H., **"A Specification-based Approach to Maintenance"**, *Journal of Software Maintenance: Research and Practice, 3*, Dec. 1991, pp. 193-213

[73] Leblang, D. B., Chase, R. P. Jr. and Spilke, H., **"Increasing Productivity with a Parallel Configuration Manager"**, *Proc. of International Workshop on Software Version and Configuration Control, Grassau*, Jan. 1988, pp. 21-37

[74] Leblang, D. B. and McLean, G. D. Jr, **"Configuration Management for Large-Scale Software Development Efforts"**, *Proc. of Workshop on Software Engineering Environments for Programming-in-the-large*, Jun. 1985, pp. 122-127

[75] Lientz, B. P. and Swanson, E. F., **Software Maintenance Management**, *Addison-Wesley*, 1980

[76] Liu, C., **"A Look at Software Maintenance"**, *Datamation, 22 (11)*, Nov. 1976, pp. 51-55

[77] Lundholm, P., **"Design Management in Base/OPEN"**, *Software Engineering Notes, 14 (7)*, Nov. 1989, pp. 38-41

[78] Madhavji, N. H., **"The Process Cycle"**, *Software Engineering Journal, 6 (5)*, Sep. 1991, pp. 234-242

[79] Mahler, A. and Lampen, A., **"An Integrated Toolset for Engineering Software Configurations"**, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium Practical Software Development Environments, Boston, MA*, Nov. 1988, pp. 191-200

[80] Martin, J. and McClure, C., **Software Maintenance - The Problem and Its Solutions**, *Prentice Hall*, 1983

[81] Munro, M., **"Software Maintenance, Reuse, and Reverse Engineering"**, *Software Reuse and Reverse Engineering in Practice, edited by D. A. V. Hall, Chapman & Hall*, 1992, pp. 573-584

[82] Patkau, B. H., **A Foundation for Software Maintenance**, *MSc Thesis, University of Toronto, Canada*, Dec. 1983

[83] Pau, L. and Kristinsson, J. B., **"SOFTM: A Software Maintenance Expert System in Prolog"**, *Journal of Software Maintenance: Research and Practice, 2* , Jun. 1990, pp. 87-111

[84] Pau, L. and Negret, J. M., **"SOFTM: A Software Maintenance Expert System in Prolog"**, *Proc. of Conference on Software Maintenance - 1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 306-311

[85] Penedo, M. H. and Shu, C., **"Acquiring Experiences with the Modelling and Implementation of the Project Life-Cycle Process: the PMDB Work"**, *Software Engineering Journal, 6 (5)*, Sep. 1991, pp. 259-274

[86] Pfleeger, S. L. and Bohner, S. A., **"A Framework for Software Maintenance Metrics"**, *Proc. of Conference on Software Maintenance-1990, San Diego, California*, Nov. 26-29, 1990, pp. 320-327

[87] Ploedereder, E. and Fergany, A., **"The Data Model of the Configuration Management Assistant"**, *Software Engineering Notes, 14 (7)*, Nov. 1989, pp. 5-14

[88] Pressman, R. S., **Software Engineering - A Practitioner's Approach, 3rd Edition**, *McGraw-Hill International Editions*, 1992

[89] Ramamoorthy, C. V., Usuda, Y., Prakash, A. and Tsai, W. T., **"The Evolution Support Environment System"**, *IEEE Transactions on Software Engineering, 16 (11)*, Nov. 1990, pp. 1225-1234

[90] Ramamoorthy, C. V., Usuda, Y., Tsai, W-T. and Prakash, A., **"GENESIS: An Integrated Environment for Developing and Evolution of Software"**, *Proc. IEEE COMPSAC*, 1985, pp. 472-479

[91] Reedy, A., Stephenson, D., Dudar, E. and Blumberg, F. C., **"Software Configuration Management Issues in the Maintenance of Ada Software Systems"**, *Proc. of Conference on Software Maintenance-1989, Miami, Florida*, October 16-19, 1989, pp. 234-245

[92] Riddle, W. E., **"Session Summary - Opening Session"**, *Proc. of the 4th International Software Process Workshop: Representing and Enacting the Software Process, published in ACM Software Engineering Notes, 14 (4)*, Jun. 1989, pp. 5-10

[93] Rochkind, M. J., **"The Source Code Control System"**, *IEEE Transactions on Software Engineering, 1 (4)*, Dec. 1975, pp. 364-370

[94] Rombach, H. D., **"Software Reuse: A Key to the Maintenance Problem"**, *Information and Software Technology*, Jan./Feb. 1991, pp. 86-92

[95] Rombach, H. D. and Mark, L., **"Software Process & Product Specifications: A Basis for Generating Customized Software Engineering Information Bases"**, *Proc. of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II - Software Track*, 1989, pp. 165-174

[96] Rose, T. and Jarke, M., **"A Decision-Based Configuration Process Model"**, *Proc. of 12th International Conference on Software Engineering, Nice, France*, 1990, pp. 316-325

[97] Rumbaugh, J., Blaha, M., Premerlani, W. Eddy, F. and Lorensen, W., **Object-Oriented Modeling and Design**, *Prentice Hall*, 1991

[98] Schneidewind, N. F., **"The State of Software Maintenance"**, *IEEE Transactions on Software Engineering, 13 (3)*, Mar. 1987, pp. 303-310

[99] Schwanke, R. W. and Platoff, M. A., **"Cross References are Features"**, *Proc. of the 2nd. International Workshop on Software Configuration Management, Princeton, New Jersey*, Oct. 24, 1989, pp. 86-95

[100] Sharpley, W. K., **"Software Maintenance Planning for Embedded Computer Systems"**, *Proc. IEEE COMPSAC 77*, Nov. 1977, pp. 520-526

[101] Shigo O., Wada, Y., Terashima, Y., Iwamoto, K., and Nishimura, T., **"Configuration Control for Evolutional Software Products"**, *Proc. of 6th International Conference on Software Engineering, Tokyo, Japan*, Sep. 1982, pp. 68-75

[102] Simmonds, I., **"Configuration Management in the PACT Software Engineering Environment"**, *Software Engineering Notes, 14 (7)*, Nov. 1989, pp. 118-121

[103] Sneed, H. M. and Jandrasics, G., **"Inverse Transformation of Software from Code to Specification"**, *Proc. of Conference on Software Maintenance-1988, Phoenix, Arizona*, Oct. 24-27, 1988, pp. 102-109

[104] Sommerville, I., **Software Engineering - Fourth Edition**, *Addison-Wesley Publishing Company*, 1992

[105] Prepared by industry with the support of the DTI and NCC, **The Starts Guide**, *2nd edition*, 1987

[106] Swanson, E. B. and Beath, C. M., **Maintaining Information Systems in Organizations**, *John Wiley & Sons*, 1987

[107] Swanson, E. B., **"The Dimensions of Maintenance"**, *Proc. of 2nd International Conference on Software Engineering, San Francisco*, 13-15, Oct. 1976, pp. 492-497

[108] Tichy, W. F., **"RCS - A System for Version Control"**, *Software Practice and Experience, 15 (7)*, Jul. 1985, pp. 637-654

[109] Tichy, W. F., **"Design, Implementation and Evaluation of a Revision Control System"**, *Proc. of 6th International Conference on Software Engineering, Tokyo, Japan*, Sep. 1982, pp. 58-67

[110] Traina Jr, C., Akhras, F. N., Capretz, L. F., Carvalho, M. B., Jino, M., and Capretz, M. A. M., **"SIPS - Current State"**, *Proc. of XIX Brazilian Conference on Informatics (SUCESU), Rio de Janeiro, RJ, Brazil, (in Portuguese)*, Aug. 1986, pp. 297-305

[111] Traina Jr, C., **Data Model and Machine Dedicated to Engineering Applications** , *PhD Thesis, IFQSC-USP, University of Sao Paulo, Sao Carlos-SP, Brazil, (in Portuguese)*, Dec. 1986

[112] Wand, Y., **"A Proposal for a Formal Model of Objects"**, *Object-Oriented Concepts, Databases, and Applications, ed. by W. Kim and F. H. Lochovsky, Addison-Wesley Publishing Company*, 1989, pp. 537-559

[113] Ward, M., Calliss, F. W. and Munro, M., **"The Maintainer's Assistant"**, *Proc. of Conference on Software Maintenance-1989, Miami, Florida*, October 16-19, 1989, pp. 307-315

[114] Yang, H., **"The Supporting Environment for a Reverse Engineering System - The Maintainer's Assistant"**, *Proc. of Conference on Software Maintenance-1991, Sorrento, Italy*, Oct. 15-17, 1991, pp. 13-22

[115] Yau, S. S. and Collofello, J. S., **"Some Stability Measures for Software Maintenance"**, *IEEE Transactions on Software Engineering, 6 (6)*, Nov. 1980, pp. 545-552