Inverse software configuration management

Rachel Jane McCrindle

Inverse Software Configuration Management

Rachel Jane McCrindle

PhD Thesis

University of Durham

Department of Computer Science

Centre for Software Maintenance

1998

Abstract

Software systems are playing an increasingly important role in almost every aspect of today's society such that they impact on our businesses, industry, leisure, health and safety. Many of these systems are extremely large and complex and depend upon the correct interaction of many hundreds or even thousands of heterogeneous components. Commensurate with this increased reliance on software is the need for high quality products that meet customer expectations, perform reliably and which can be cost-effectively and safely maintained. Techniques such as software configuration management have proved to be invaluable during the development process to ensure that this is the case. However, there are a very large number of legacy systems which were not developed under controlled conditions, but which still need to be maintained due to the heavy investment incorporated within them. Such systems are characterised by extremely high program comprehension overheads and the probability that new errors will be introduced during the maintenance process often with serious consequences.

To address the issues concerning maintenance of legacy systems this thesis has defined and developed a new process and associated maintenance model, Inverse Software Configuration Management (ISCM). This model centres on a layered approach to the program comprehension process through the definition of a number of software configuration abstractions. This information together with the set of rules for reclaiming the information is stored within an Extensible System Information Base (ESIB) via, the definition of a Programming-in-the-Environment (PITE) language, the Inverse Configuration Description Language (ICDL). In order to assist the application of the ISCM process across a wide range of software applications and system architectures, the PISCES (Proforma Identification Scheme for Configurations of Existing Systems) method has been developed as a series of defined procedures and guidelines. To underpin the method and to offer a user-friendly interface to the process a series of templates, the Proforma Increasing Complexity Series (PICS) has been developed.

To enable the useful employment of these techniques on large-scale systems, the subject of automation has been addressed through the development of a flexible meta-CASE environment, the PISCES M^4 (MultiMedia Maintenance Manager) system. Of particular interest within this environment is the provision of a multimedia user interface (MUI) to the maintenance process. As a means of evaluating the PISCES method and to provide feedback into the ISCM process a number of practical applications have been modelled.

In summary, this research has considered a number of concepts some of which are innovative in themselves, others of which are used in an innovative manner. In combination these concepts may be considered to considerably advance the knowledge and understanding of the comprehension process during the maintenance of legacy software systems. A number of publications have already resulted from the research and several more are in preparation. Additionally a number of areas for further study have been identified some of which are already underway as funded research and development projects.

This thesis is dedicated to:

My husband, Fred
My parents, Margaret and David, and brother, 'J'
My mother-in-law, Mary
and to the memory of my father-in-law, Arthur.

Acknowledgements

I would like to thank all those who have helped me complete this work. In particular I would like to thank my supervisor, Malcolm Munro, for his help and guidance throughout the duration of the research. I would also like to thank John Foster and Richard Warden for their assistance in defining an initial area of study. I would like to thank Stuart Doggett and Kirsty-Anne Dempsy, final year project students and Frode Sandnes, Research Assistant on the VES-GI project for their help in implementing some of the ideas stemming from the research work. Lastly, I would especially like to thank my husband, Fred, for his proof reading skills, for his patience and for always being there for me.

Statement of Copyright

For so it falls out
That what we have, we prize not to the worth
Whiles we enjoy it, but, being lacked and lost,
Why then we rack the value, then we find
The virtue that possession would not show us
Whiles it was ours

Shakespeare Much Ado About Nothing, Act IV, Scene I

Table of Contents

Chapter 4 Inverse Software Configuration Management

Chapter 7 Critical Evaluation & Discussion of the Research Objectives

Chapter 8 Conclusions & Further Work

Figures

Tables

Chapter 1

Introduction

Proceed, proceed. We'll so begin these rites
As we do trust they'll end in true delights

Shakespeare As You Like It Act V, Scene IV

1.1 Introduction

The last decade has witnessed an explosion in terms of the abundance, size and complexity of software systems being developed such that they now play a key role in almost every aspect of today's society [167, 169, 241, 306, 322]. Indeed, the software industry may be seen as one of continual growth: in the industrial sector as automatic control of many systems, previously unrealisable due to hardware limitations, becomes possible through the transference of complex algorithms to a software base [392]; and in the business arena where software plays an increasingly central role in business process optimisation and redesign [6, 163]. Commensurate with the increasing reliance being placed on software is the need for cost effective, high quality software products that meet customer expectations, perform reliably and safely [44, 325], and which can evolve to meet the changing requirements of a dynamic population [241]. This expansion of the software industry has brought about two significant changes in the way that software maintenance is perceived. Firstly, it has emerged as a discipline in its own right within the software engineering field. Secondly, it has been integrated into the software lifecycle such that maintenance is no longer considered to be purely an ad-hoc extension to the development process. Additionally, recognition of software as a key corporate asset is now becoming evident and the importance of maintaining this asset is gaining momentum [73, 97, 163, 306]. The importance of investing in the maintenance and control of software is further substantiated by the fact that in many of today's computer systems the hardware expenditure can no longer be considered to be the major factor in any costing scenario [170, 348, 242].

This growing awareness of the importance of software has precipitated an increase in the research being conducted within the software maintenance arena. This research falls into two main categories: that of investigating methods for developing new software systems with enhanced future maintainability, and that of finding ways to more effectively maintain operational or legacy systems. This thesis is concerned primarily with the latter.

1.2 Purpose of the Research

Unless an operational software system has been developed and maintained under strict configuration control it is likely that the cost of maintenance will be very high. Indeed surveys have shown that software maintenance may account for between 40 and 85 percent of a company's software expenditure per annum [10, 49, 165, 243, 405]. This high cost may be largely attributed to the time spent identifying and understanding the parts of a system potentially affected by proposed maintenance changes [306]. In large legacy systems the process of identification and understanding is greatly compounded by several time-related factors which act individually or in combination to precipitate a '*loss of control*' of the system. These factors include [262]:

- increased component and relationship complexity over time,
- system version explosions,
- lack or loss of useful documentation,
- personnel changes, and
- maintenance induced errors.

If legacy systems are to be effectively maintained there is a need to regain control of the system and to minimise the time spent trying to understand and identify the parts of the system affected by a change. Software Configuration Management (SCM) is an already recognised discipline for controlling the evolution of complex software systems with the effect of successfully reducing costs for systems under development [65]. This discipline may also be applied during the maintenance process if a software/system configuration is already known, however this is rarely the case in legacy systems due to the reasons given above. Therefore in order to address '*out of control*' legacy systems there is a requirement to re-identify the existent but often corrupted configuration together with its associated dependencies. **This thesis defines a new process and maintenance model, Inverse Software Configuration Management (ISCM), and an associated method, the Proforma Identification Scheme for Configurations of Existing Systems (PISCES), which enables existing (legacy) software systems to be returned to configuration control.**

1.3 Research Outline

The ISCM process is primarily concerned with identifying and documenting the current configurations or software system architectures of legacy systems. Additionally, any knowledge or understanding gained in relation to the function of the affected parts of the system is also essential and must be recorded as part of the ISCM process. This knowledge arises from a combination of code analysis, associated documentation, domain knowledge [60, 234] and often a large proportion of detective work [46]. The entire process may be termed system comprehension and encompasses understanding at both the program and system levels. Successful configuration identification and knowledge elicitation will enable subsequent modifications to the system to be made in a more efficient, controlled and cost effective manner. The ultimate goal of the ISCM process is to enable future maintenance to proceed as if the system was developed using SCM principles from the outset. Alternatively, if the structure of a legacy system has degraded to such as extent that it is in danger of being *'maintained to death'*, ISCM provides a solid foundation for the re-engineering of part or all of the system.

In order to drive the ISCM process, information about software system configurations must be systematically gathered and persistently stored. To meet this requirement the framework for an ***Extensible System Information Base (ESIB)*** has been defined. The ESIB contains information, rules and data related to numerous software system architectures and operating environments. After initial seeding, the information base evolves incrementally such that the greater number of maintenance projects it encompasses the more accurate and more automated the ISCM process becomes. In order to seed the information base this research investigated what constitutes a software system and hence what components need to be identified, traced and documented in order to reconstitute the software configuration description. This was achieved through a manual study of two UNIX applications, RCS (Revision Control System) and SPMS (Software Project Management System). To complement the study an approach to ***modelling abstractions*** of software system architectures was developed and a ***language*** defined for syntactically and semantically describing the resultant software configurations.

A key objective of the research is the meaningful representation of the information held in the knowledge base. This is achieved through a series of abstractions of the system configuration documented via generic, tailored and specific ***natural language proformas***. The benefits of such an approach are threefold. Firstly, it increases the genericity of the ISCM model across operating platforms, application system types and languages. Secondly, it enables the presentation of a number of different views of the system at a level appropriate to clients, managers and maintainers, whilst preserving comprehensibility and rigour. Thirdly, by strictly defining the structure of the proformas and by providing guidelines for their completion through the levels of abstraction, a consistent and traceable set of documentation can be produced, enabling a more methodical approach to the maintenance process to be established.

Identification of the system configuration and its associated dependencies greatly enhances the understanding of a system about to be maintained. Combining all these factors to obtain system understanding is often very time consuming and is generally repeated each time the same area of code is maintained. Therefore, if significant savings in the cost of maintenance are to be realised there is a requirement to preserve the understanding of the maintainer such that they, and in particular different maintainers, do not have to go through the same process each time a system is to be modified. ISCM therefore places emphasis on the ***incremental documentation*** and updating of the system understanding recovered during the maintenance process.

In order to establish the role of the ***ISCM process*** within the global framework of software maintenance an ***ISCM maintenance process model*** has been defined. This process model represents the series of activities that occur from the inception of maintenance, through the system comprehension process to completion of the required modification. To assist in the production of the system architecture descriptions and to help standardise the ISCM process, an associated method, ***Proforma Identification Scheme for Configurations of Existing Systems (PISCES)***, has been developed. The method comprises a logical sequence of steps through each of the activities defined in the process model until the system architecture reconstructions have been successfully determined. The PISCES method is centred around development of a series of pre-defined proformas which gradually become more complex and descriptive as they are incrementally populated with system information.

An associated prototype meta-CASE (Computer Aided Software Engineering) environment, the ***PISCES M^4 (MultiMedia Maintenance Manager) system***, has been developed to assist in the implementation of the PISCES method. The PISCES M^4 system thereby semi-automates the ISCM process. Essentially the system consists of a set of tools linked together through the M^4 interface. At the centre of the system is the PISCES tool which consists of a series of form-fill interfaces which in the first instance prompt the maintainer for background information about the system to be maintained. As a result of the data provided by the maintainer, details about the extraction tools available for configuration retrieval are returned to the maintainer, along with any pre-existing information regarding the function of that area of the system. The maintainer may then choose to evoke these tools, which reside on the application system host environment, and download the resultant information onto the PC (Personal Computer) mounted PISCES M^4 system. Alternatively, the maintainer may choose to browse the information already reclaimed during previous comprehension activities for a particular system. The PISCES M^4 system then acts as a sophisticated report generator producing visual representations of the system configuration, domain knowledge descriptions etc. The initial prototype tool utilised hypertext technology [99, 265] but more recent research has developed a hypermedia interface to the system comprehension process [291, 267, 268, 269, 270].

One of the key features of the PISCES M^4 system is the use it makes of the operational environment of the application system. One of the common criticisms of CASE tools is the high initial investment that has to be committed prior to any tangible gains. This cost often acts as a barrier to the widespread uptake of CASE technology [200]. With this in mind the PISCES M^4 system utilises, as far as possible, tools that already exist on the host computer, for extraction of information regarding the system architecture descriptions. In this way automated support can be delivered cost effectively. The primary roles of the PISCES M^4 system are therefore to act as a front-end to the program comprehension process and as a specialised form of report co-ordinator and generator for the recovered information.

The combination of the extraction of dependency information, its visual representation, the PISCES method and incremental documentation of system understanding reduce the financial burden of program and system comprehension during software maintenance. It has been estimated that the world cost of software development in 1995 amounted to $500 billion [359]. More recently, figures of £400 billion are being quoted as the additional world-wide cost of solving the *'Year 2000'* problem [7, 406], representing approximately 20% of every commercial organisation's IT budget [54]. Hence even if the ISCM process only realises a 1% reduction in the amount of effort expended on maintenance, potentially £4 billion on maintenance could be saved, thereby substantially increasing the investment available for new software development [16].

1.4 Thesis Aims and Objectives

The primary contribution of this research is the definition of a new process and maintenance model, ***Inverse Software Configuration Management*** (ISCM), and its associated method the ***Proforma Identification Scheme for Configurations of Existing Systems*** (PISCES). Surveys of the literature do not reveal other work that tackles the SCM process predominately from the inverse perspective, i.e. that of bringing *'out of control'* systems back under configuration control. The originality of the ISCM approach also lies in its unique combination of a number of proven techniques in order to produce a low cost, generic, understandable, yet rigorous solution to one of the key problems of software maintenance today, that of program and system comprehension.

Additionally, no other systems currently offer a ***Multimedia User Interface*** (MUI) to the representation of knowledge reclaimed during the program comprehension process.

In order to define and describe the ISCM process, ISCM maintenance model and associated PISCES method a number of key objectives have been identified:

- ***Objective 1:*** Definition and development of the ISCM process and its associated maintenance process model.
- ***Objective 2:*** Description, guidelines and documentation for each phase of the PISCES method.
- ***Objective 3:*** Modelling of software system architectures through the process of abstraction.
- ***Objective 4:*** Definition and structure of the underpinning ESIB.
- ***Objective 5:*** Definition of the Inverse Configuration Description Language (ICDL) for syntactically and semantically describing software system configurations at a programming-in-the-environment (PITE) level.
- ***Objective 6:*** Definition of the natural language representation of the model via the Proforma Increasing Complexity Series (PICS) of templates at the generic, specific and tailored levels of abstraction.
- ***Objective 7:*** Implementation of the PISCES M^4 prototype system and demonstration of semi-automated support for the ISCM process and PISCES method.
- ***Objective 8:*** Practical application of the ISCM process, ISCM maintenance model, PISCES method and M^4 system in the maintenance of a number of 'real-world' systems.
- ***Objective 9:*** Critical evaluation of the ISCM process, ISCM maintenance model, PISCES method and PISCES M^4 system.
- ***Objective 10:*** Identification of future extensions to the work in the area of Inverse Software Configuration Management and in associated fields of study.

1.5 Thesis Overview

This section outlines the material that will be covered within each chapter of the thesis. A thesis road map linking chapter numbers to the objectives set in section 1.4 may also be found in Figure C1-1 at the end of this chapter.

- ***Chapter 2 - The Software Maintenance and Software Configuration Management Processes:*** this chapter provides the background material for the thesis which, due to the 'activity-combining' nature of the Inverse Software Configuration Management (ISCM) process, necessitates treatment of a number of different areas. Section 2.2 introduces the main activities of the software maintenance process and explains why software maintenance is a persistent and difficult problem. It also considers the key problem of program and system comprehension on which much of this thesis is based. Section 2.3 defines the Software Configuration Management (SCM) discipline and discusses the notions, concepts and characteristics that underpin the ISCM process. It also describes a number of currently available utilities, tools and systems that can be used to control software system configurations. Section 2.4 addresses the key problems associated with identifying software system configurations in large legacy systems and how these relate to software maintenance and software configuration management. Section 2.5 introduces a number of other related activities, namely reverse engineering, re-engineering, redocumentation and restructuring while section 2.6 discusses the relationships existing amongst these activities, the software maintenance process and the software configuration management discipline. Section 2.7 explains the importance of documenting all of the knowledge recovered during the software maintenance process and examines some of the currently available methods for achieving this. Sections 2.8 and 2.9 conclude the chapter with a summary of the key concepts introduced and by highlighting the contribution to be made in these areas by the ISCM process, its associated model and the PISCES method.

- ***Chapter 3 – Managing & Modelling Software Configurations:*** this chapter describes in more depth the key issues that impact upon the ability of a maintainer to manage and model software system configurations. Section 3.2 outlines a number of existing process models for software development and maintenance and highlights the issues that remain to be addressed in future models. Section 3.3 examines the factors prevalent in modern computer systems that impact on the level of complexity of software systems. It also discusses what constitutes the basic building blocks of a software configuration, and it critically reviews a number of current methods for describing such configurations.

Section 3.4 outlines the process whereby software systems are composed from their constituent parts during the traditional SCM process, with a view to identifying key activities for use when reclaiming 'lost' legacy configurations. The chapter concludes by discussing a number of methods by which software system configurations can be stored and organised.

- ***Chapter 4 - The Inverse Software Configuration Management Process:*** this chapter forms the core of the thesis and presents the original concepts developed during the research. Section 4.2 proposes and defines a new process Inverse Software Configuration Management (ISCM) and expands upon the notions, concepts and characteristics of the Inverse Software Configuration Identification (ISCI) phase of the process which forms the foundation of the ISCM model. Section 4.3 describes the component stages of the ISCM process model and explains the role played by each stage in the reclamation of configurations of legacy systems. Section 4.4 describes more specifically the modelling of software system architectures within the context of the ISCM process by detailing the different component types that constitute a system configuration, the relationships existing between them and the levels of abstraction at which they may be viewed and understood. Sections 4.5 and 4.6 develop the Inverse Configuration Description Language (ICDL) and the Proforma Increasing Complexity Series (PICS) as machine-readable and user-friendly representations respectively, for describing legacy system configurations at a programming-in-the-environment (PITE) level. Section 4.7 justifies in relation to the traditional SCM building process a number of key activities that need to be addressed if the proformas are to be progressively populated during maintenance and 'lost' legacy configurations reclaimed. The chapter concludes by discussing how the ISCM discipline extends the SCM discipline to assist in the maintenance of large legacy systems.

- ***Chapter 5 - PISCES Prototype System:*** this chapter describes the implementation of the ISCM process via the PISCES (Proforma Identification Scheme for Configurations of Existing Systems) prototypes. Section 5.2 describes the overall architecture of the PISCES tool while section 5.3 briefly describes how this architecture was implemented in each of the four prototypes developed during the course of the research. Section 5.4 describes the functionality offered by each of the bespoke tools developed for the PISCES M^4 system. The chapter concludes by describing how the PISCES M^4 system can be used to realise the ISCM process.

- ***Chapter 6 - Application:*** this chapter provides an account of how the ISCM method and PISCES tool relate to real-world applications. Section 6.2 describes four trial applications of increasing size and complexity plus the PISCES M^4 system itself. Section 6.3 describes on a process by process basis the application of the ISCM process model, PISCES method and M^4 system to the maintenance of legacy software systems. The extent to which the modelling process varies according to the size and complexity of the system being maintained is also discussed. The chapter concludes with an assessment of the applicability and effectiveness of the ISCM process and PISCES method to the program comprehension process.

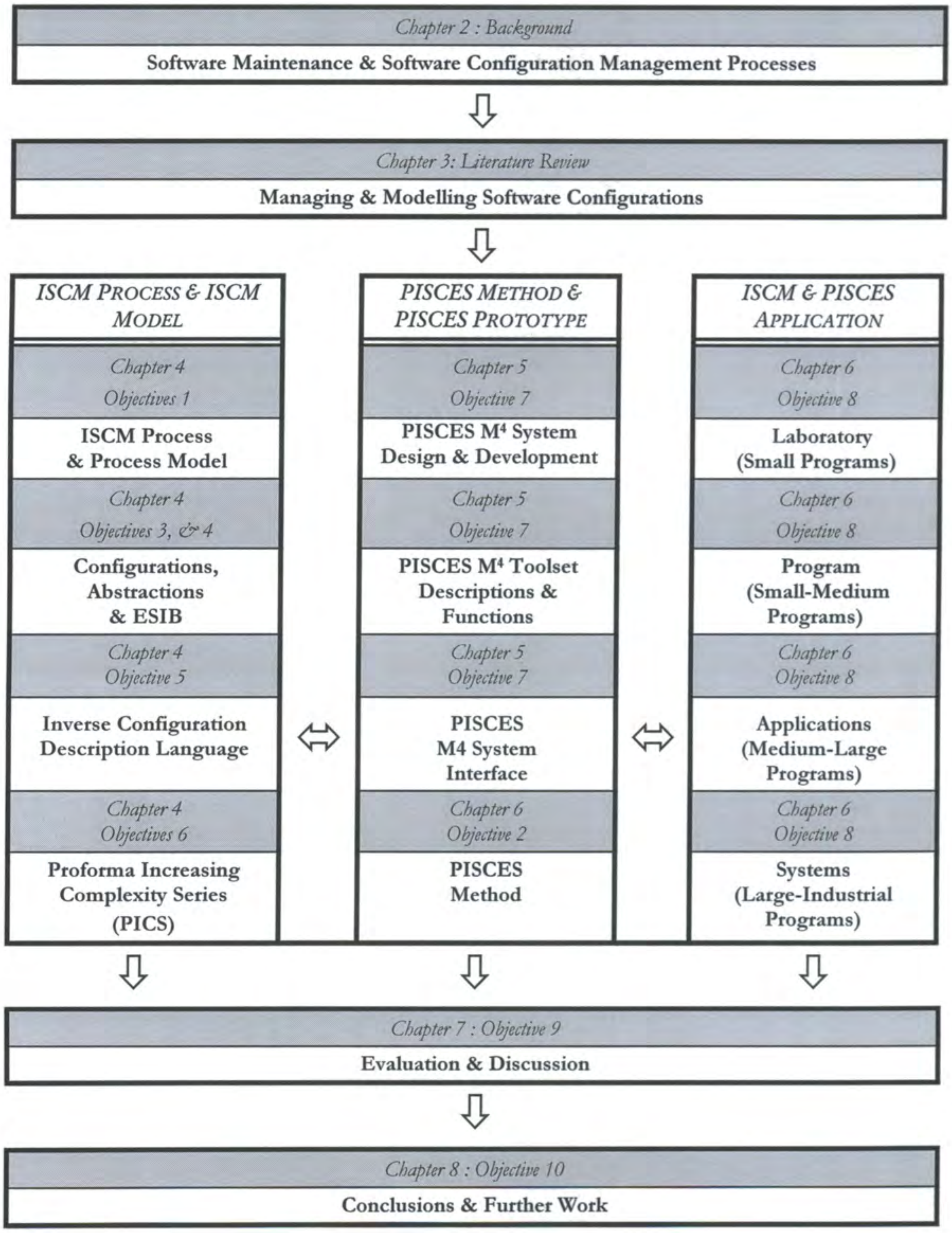
- ***Chapter 7 - Evaluation and Discussion of Objectives Set:*** this chapter evaluates and discusses the ISCM process and model and the associated PISCES method and tool. Section 7.2 critically reviews the process, method and associated tool in relation to the objectives laid down in section 1.4 and improvements are suggested where appropriate. The chapter concludes with an overall assessment of the degree to which the objectives set within the thesis have been achieved and meet the needs of a modern industry.

- ***Chapter 8 - Conclusions and Further Work:*** this chapter reviews and draws together the research undertaken. Section 8.2 discusses the current state of software maintenance in general. Section 8.3 summarises the subject of the research, presents the major findings of the earlier chapters and formulates conclusions about the success of the work. Section 8.4 suggests further work that can be carried out and future directions for the research. The thesis concludes by defining the role of the ISCM process and PISCES method in the field of software maintenance.

1.6 Summary

The importance of and reliance placed on software in all sectors of society has risen rapidly over the past decade. Much investment has consequently been placed in software in terms of monetary value, business credibility and often human safety [244, 307, 324, 325]. There is therefore an overwhelming requirement to cost effectively and safely maintain software systems over their lifetimes, which may be in excess of fifteen to twenty years. Without adequate control during development or maintenance software systems can very often become *'out of control'*. This has a twofold effect, program comprehension effort is very much increased and the degree of confidence with which changes may be made to a software system without adversely affecting other parts of the system is vastly reduced. A method is therefore needed to bring out of control systems back under configuration control. This thesis proposes such a process and model,

Inverse Software Configuration Management (ISCM), and its utilisation through the ***Proforma Identification Scheme for Configurations of Existing Systems*** (PISCES) method and system.



Chapter 2 : Background
Software Maintenance & Software Configuration Management Processes
Chapter 3: Literature Review
Managing & Modelling Software Configurations
ISCM PROCESS & ISCM MODEL
Chapter 4 Objectives 1
ISCM Process & Process Model
Chapter 4 Objectives 3, & 4
Configurations, Abstractions & ESIB
Chapter 4 Objective 5
Inverse Configuration Description Language
Chapter 4 Objectives 6
Proforma Increasing Complexity Series (PICS)
PISCES METHOD & PISCES PROTOTYPE
Chapter 5 Objective 7
PISCES M4 System Design & Development
Chapter 5 Objective 7
PISCES M4 Toolset Descriptions & Functions
Chapter 5 Objective 7
PISCES M4 System Interface
Chapter 6 Objective 2
PISCES Method
ISCM & PISCES APPLICATION
Chapter 6 Objective 8
Laboratory (Small Programs)
Chapter 6 Objective 8
Program (Small-Medium Programs)
Chapter 6 Objective 8
Applications (Medium-Large Programs)
Chapter 6 Objective 8
Systems (Large-Industrial Programs)
Chapter 7 : Objective 9
Evaluation & Discussion
Chapter 8 : Objective 10
Conclusions & Further Work

Figure C1-1 Thesis road map

Chapter 2

The Software Maintenance and Software Configuration Management Processes

Not what is dangerous present, but the loss
Of what is past

Shakespeare Coriolanis, Act III, Scene II

2.1 Introduction

This chapter presents the concepts that underpin the Inverse Software Configuration Management (ISCM) process. In particular the chapter outlines the software maintenance and software configuration management (SCM) processes and discusses the relationships that exist between them. It also introduces the key software maintenance problem of program and system comprehension, and discusses how effective usage of SCM techniques can reduce the comprehension overhead. The chapter also establishes the foundation for how the ISCM process at the centre of this thesis may be integrated into the forward and reverse engineered software system lifecycles. A review of current work relevant to the research topics is also undertaken. The chapter concludes by outlining a number of areas, regarding maintenance and configuration management principles and practices, which require further work if the identified problems are to be solved and legacy software systems brought back under configuration control.

2.2 Software Maintenance

Since the genesis of software development many definitions of software maintenance have been put forward [109, 142, 151, 243, 300]. These range from the simplistic and outdated view of maintenance as purely a 'bug' fixing exercise, to its role as a complex combination of different activities encompassing enhancements, amendments and fixes. Lately, the recognition and definition of software maintenance as a discipline in its own right has lead to a far more uniform usage of the term '*software maintenance*'. Software maintenance may thus be defined as:

> "*the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment*" [198].

That is, software maintenance may be said to encompass all activities that take place after a software product has been delivered to the customer. This emerging '*de facto*' definition of software maintenance is however commonly further refined into four maintenance sub-disciplines, which more closely define and distinguish between the nature of the different software maintenance activities:

- *Corrective Maintenance:* is concerned with making changes to code that result from inconsistencies between the specification of the product and the product itself [183]. It is a reactive process [257] to correct faults and omissions, and primarily involves changes to the code itself. Corrective maintenance is the 'traditional' view of software maintenance and yet only accounts for approximately 17% of total maintenance expenditure.

- *Adaptive Maintenance:* is concerned with making changes to both the design and code in order to enable the software product to accommodate changes in its hardware and software operating environments [61, 184, 257]. This type of maintenance accounts for approximately 18% of total maintenance effort.

- *Perfective Maintenance:* is concerned with improving the function of the software in response to user requests for improvement, enhancements to functionality, or customisation to new working practices. It also encompasses changes in the processing environment due to changes in rules, laws and regulations. It involves changes to the specification, design and code and accounts for 60% of maintenance effort. This high percentage reflects industrial statistics that show that around 10% of a software system will change each year in response to modifications to the user requirements [35].

- ***Preventive Maintenance:*** is concerned with updating software in order to preclude future problems and facilitate future maintenance work [34, 183]. Although it accounts for only 5% of the maintenance effort and is largely associated with legacy systems it still accounts for a significant proportion of effort when considered in monetary terms and also in terms of the savings which could accrue if preventive maintenance is carried out effectively. It is proactive in nature and has very close links with the re-engineering aspect of the software lifecycle.

In addition some authors consider that the definition of software maintenance should be extended to include ***user support*** and ***training*** of end users as well as actual changes and updates to the product itself [10, 151, 183, 233]. McCrindle has also proposed the term '***pre-emptive***' maintenance to embody activities during development whose primary aim is to reduce future maintenance costs [267].

2.2.1 Software Maintenance as a Persistent Problem

The myths of software maintenance as being purely a 'bug fixing' exercise have now been largely overcome. This has resulted in expenditure on software maintenance no longer being seen as a failing of the development team and the company as a whole. Indeed, Lehman has observed and stated in his law of continuing change, that a program in a real-world environment must change or become progressively less useful in that environment [239, 240, 241], a view concurred with by Parnas [306]. Correspondingly, technical personnel and senior management are now more inclined to talk about the real cost of maintenance within their organisations and to discuss their specific maintenance needs. This, together with studies into software maintenance [33, 119, 189, 201, 233, 234, 405] is enabling the maintenance community to gain a more accurate profile of the spectrum of maintenance activities; a more realistic estimate of the real cost of software maintenance; and most importantly, indicators of areas that must be addressed through research and development programmes [98]. The acceptance of software as an asset and maintenance as a necessary discipline is visible through its much higher profile than a decade ago [85]. For example, there are now an increasing number of national, European and international maintenance workshops and conferences, specialist maintenance journals, references to maintenance within the popular and computing press and a proliferation of papers addressing maintenance related issues in most computing journals.

However recognition must not become an excuse for complacency. The costs of maintenance also serve as indicators that companies cannot hide behind the activities of software maintenance as justification for additional software expenditure. Indeed, the cost of software maintenance as a proportion of the software budget of a company has increased steadily during the past 20 years, rising from 35-40% in the 1970s to 60% in the 1980s and, is predicted to rise to more than 80% by the end of the 1990s if the maintenance approach is not improved [315]. Indeed, as well as the capital costs incurred, other problems arise due to the 'maintenance burden' such as lost market development opportunity [306], customer dissatisfaction and lower quality software [315, 350] which cumulatively could prove fatal to some organisations. Emphasis must therefore be placed on performing the most cost effective maintenance possible, whatever the type of maintenance required.

That maintenance costs are continuing to rise, may in some aspects be considered a surprising concept considering the advances made in software technology [322], analysis and design methods, programming languages, computer aided software engineering (CASE) tools and project management since the software crisis of the late 1970s and early 80s. However, increasing figures may be attributed to a number of factors such as the increasing proliferation of software into previously non-computerised disciplines; the increasing complexity of software as its limits are extended [393]; the increasing distributed and embedded nature of software [19, 345, 386]; and ultimately the ability to 'create chaos more quickly' with the plethora of new CASE tools and rapid application development (RAD) environments. Software is also unique in that within a person's lifetime, a software system can go through up to eight technological revolutions compared to only two in most other engineering disciplines, such that the 'unthinkable' becomes possible within twenty years [370, 406]. For example, the lunar module that first landed on the moon had less computing power than a Ford car of today and the astronauts within that module had never seen a microchip or pocket calculator [370]. Additionally, software products traverse a wide range of disciplines such that they incorporate art, science, craft, fire-fighting and archaeological discovery skills as well as those of an engineering nature [261]. It must also be remembered that each new software system of today is a legacy system of tomorrow requiring maintenance. Indeed, even the most successful software systems require maintenance as companies strive to retain their asset for as long as is economically possible [36]. Companies are therefore bound to adapt and enhance their systems to take into account upgrades in their operating environments, working practices and even movements from one millennium to the next [105, 284]. This is evident as a 10% growth in system size per year [365], a corresponding requirement for a 15% annual increase in programming staff just to meet the increased maintenance needs [209], and a resultant estimated shortfall of some 300,000 software professionals to satisfy this need [406].

2.2.2 Maintenance vs. Development

Some practitioners view maintenance as further development. Indeed this may be a valid assumption if a product has been developed with maintenance in mind from the outset of development. However in general, maintenance, and in particular maintenance of legacy systems is far more complex than development of green-field software systems. The reasons for this include:

- ***Protracted timescales***: development projects are undertaken within a timescale and to a predefined budget at the end of which they produce an identifiable and specified product [34]. Conversely, maintenance projects are normally open ended and may continue for many years, the objective being to extend the life of a piece of software for as long as is economically possible.

- ***Masked evolution***: it is often difficult or impossible to trace the evolution of a legacy software system through its many versions or releases. Similarly, it is also generally difficult or impossible to trace the process through which these versions and releases were created.

- ***Comprehension difficulties***: it is often exceptionally difficult to understand software that was written by another programmer or team. This problem is often accentuated by the probability that the original programmer is no longer available for consultation [74]. Additionally there is often no recorded documentation [405], or the documentation that does exist no longer represents accurately the status of the system.

- ***Change inflexibility***: most legacy software has not been designed for change and it is therefore extremely difficult to incorporate even very small changes into the code without 'firing off' the ripple effect [151]. This leads to potentially more errors being introduced into a piece of software than have been corrected.

- ***No choice acceptance***: software to be maintained must be accepted as it is. In contrast, a client prior to delivery may reject unsatisfactory development projects if they are of poor quality or are insufficiently documented. Legacy systems must be accepted as they are, even if they have become extremely degraded and out of control.

- ***Poor image***: socially and technically, maintenance to date has not been viewed as glamorous work and has therefore not always received the recognition it deserves. Allied to this, capital investment in the maintenance of software as an asset has generally been low. However a shift is occurring as software maintenance managers learn to speak the 'language of business' [97, 234] and senior managers accept maintenance as adding value to a product [168].

- ***Immature discipline***: the maintenance discipline is still in its infancy and hence the number of methods, tools and techniques are still lagging behind those of development although this situation is now improving [52, 53, 405].

Whilst all of the above factors contribute to the difficulties of maintenance, there is one intrinsic problem at the centre of all software systems that has to be overcome before 'safe' maintenance of legacy systems can take place. This key problem facing maintainers of existing or legacy systems is that of program and system comprehension which may consume from 50% to 90% of maintenance time [122, 401]. Thus, if the problems of comprehension can be alleviated, even by a small degree, then there is the potential for very significant maintenance savings to be made.

2.2.3 Program and system comprehension

For safe and effective maintenance to take place it is essential to understand the software product as a whole and the parts of the programs affected by a change in particular [63, 108, 177]. The process of program and system comprehension involves three key areas [368]:

- Understanding what the system does and how it relates to its environment.
- Identifying where the system change should be effected.
- Deducing how the components identified for correction or modification work.

That is, program comprehension is concerned with understanding the application system prior to making a modification. A person is said to understand a program when able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its environment in human-oriented terms rather than in the terminology and syntax of its program representation. That is, a natural language understanding is reached regarding the association between the unfamiliar program or part of the program and the formal representation of the code. This is known as the concept assignment problem [46].

There are two basic levels at which this may be achieved. The whole program may be studied and understood in its entirety, or the affected parts of the program only may be studied and understood as required [52, 245]. Given the size and complexity of today's software systems the latter option is generally considered to be the only real alternative. Indeed this view is supported by the fact that for many software systems the 80:20 rule applies, that is, 80% of maintenance activity is spent on 20% of the code.

Biggerstaff describes program comprehension as a multifaceted process of analysis, experimentation, guessing and crossword puzzle-like assembly [46]. He suggests that this can only be fully achieved by studying domain concepts and realisations as well as the program structure and functions. Brooks in an earlier study emphasised the iterative nature of the program comprehension process. He also put forward a theory based on mapping knowledge of the problem domain to the program domain, laying particular emphasis on the need to maintain a history of the decisions made during requirements analysis [60]. Other studies have also revealed the crucial importance of domain knowledge for the effective construction and maintenance of large complex systems [1, 52, 214, 348, 379, 325].

At the extremes, there are two opposing views regarding the most effective approach to program understanding, those of top-down comprehension [358] versus those of bottom-up comprehension [309]. The top-down approach is generally considered to be most appropriate for familiar programs having available domain knowledge whilst the bottom-up approach is best suited when code is completely unfamiliar to the maintainer. However, in practice some combination of the two approaches occurs [75, 88, 387, 388, 389]. That it is vital to comprehend a program before modification takes place is however not in dispute, whatever the category of maintenance required. The problem of program and system comprehension can be very large and the associated costs of understanding the system and making a 'safe' change often prohibitive to the future maintenance of a software product [340]. This fact is substantiated by reported costs for the comprehension process during maintenance, for example costs may amount to $200 million per year [303], and 50% of total maintenance effort expended [108]. The comprehension part of maintenance whilst essential may be considered to be 'unproductive' maintenance effort [271] whilst the process of actually effecting the change may be considered as productive maintenance time since it is during this period that *value* is added to the software product.

Many of the problems of controlling software development and its subsequent comprehension can however be alleviated through the use of software configuration management procedures and practices.

2.3 Software Configuration Management

Software Configuration Management (SCM) is now a recognised discipline for controlling the evolution of complex software systems [376]. It is acknowledged as adding considerable value to the control of software development [138] such that reliable software is produced that meets original design specifications on time and within budget. However, like many other key activities within software engineering, this view has taken time to evolve and adoptions of the techniques have parallels in older engineering disciplines. Indeed, the development of Configuration Management (CM) as a discipline can be traced to the hard engineering disciplines such as electro-mechanical engineering. These disciplines use change control techniques to manage blueprints and other design documents [22] and to specify how particular versions of specific components are combined to produce the final assembly [237]. These principles have been refined and tailored, initially to assist in the control of computer hardware production, and subsequently to assist in the development, production and maintenance of computer software. Configuration management traditionally treated each hardware component of a system as a separate entity, yet irrespective of size and complexity only considered the software as a single entity. It is this superficial treatment of software that has been responsible for much of the lack of visibility and corresponding lack of management control often evident in the software aspects of the project.

Additionally, attempts to directly emulate hardware configuration management techniques in the software environment have often failed, because although the principles behind hardware configuration management are directly applicable, it does not follow that their implementation is directly transferable. Indeed, there are a number of fundamental differences between the hardware and software components of a computer system, for example:

- Software is more dynamic than hardware and therefore needs to change more rapidly [348].
- Software is easier to change than hardware and therefore can change more rapidly [2, 168].
- Software progress, completion and correction is physically less evident and 'concrete' than hardware.
- Emphasis must be placed on the software deliverables and not just on the process of scheduling and managing the tasks to achieve an acceptable finished product [55].
- Emphasis must also be placed on the software that supports the production of the software end products as well as the product itself [65].

- A finished software product may be distributed over more than one site [239], whilst a hardware product once manufactured is normally bound together within a single unit.
- Software configuration management has greater potential for automation than hardware configuration management.

Additionally, the increasing amount of 'firmware', defined as a hardware device and the software that resides on that device, where the software cannot be readily modified under program control, means that both the hardware and software practices and procedures must increasingly be considered in parallel. [65].

Advances in computer science have had two very profound effects on the configuration management discipline. Firstly, until recently SCM was very much a manual set of procedures. This approach was acceptable whilst computer systems were relatively small, but with the advent of larger and more complex systems it became apparent that manual techniques could not keep pace with the speed at which changes take place or with the volumes of data produced. Manual systems became cumbersome to use and easily corrupted. Barr & Stroud found that manual methods '*did not work in practice*' because they required '*perfect people*', were difficult to prove, difficult to use and generated a vast amount of paperwork which required its own control strategy [102].

With the introduction of automated procedures greater control over factors such as who makes changes, where changes are effected and the general integrity of the resultant information could be made. The potential to automate almost all of the SCM process now exists, for example, identification, change tracking, version selection and baselining, software manufacture and simultaneous updates. Indeed, looking at the size of software projects being developed today, automation in almost every aspect of the configuration management process is becoming essential [329].

Secondly, SCM was originally considered to be a purely management discipline but this has recently progressed into the software development arena. This has been facilitated by the increasing interest in integrated programming support environments (ipses). It appears, however, that a balance must be struck between the management-oriented approach to SCM, which provides project management information and control (release management, change control procedures etc.), often seen by programmers as unnecessary red tape, and development-oriented SCM which supports the programmer in developmental tasks but provides little management information. An attempt to unify these two approaches in the form of an integrated toolset has been developed by Mahler and Lampen [253]. In an attempt to avoid confusion between SCM in a management-oriented context and development-oriented context Mahler and Lampen have used the term Software Configuration Engineering. This emphasises the more technical aspects of SCM such as version control, configuration identification and system building.

Software configuration management had been variously defined [41, 57, 190, 237, 260] ranging from simple definitions of SCM as a *technique for controlling the evolution of complex software systems* to those which are far more encompassing, for example:

> *"SCM is the process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system lifecycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items"* [17].

This IEEE (Institute of Electrical and Electronic Engineers) [17] definition shows that the discipline of Software Configuration Management (SCM) may be better described as an integrated set of four sub-disciplines: software configuration identification; software configuration control; software configuration status accounting and software configuration audit, which combine with the common goal of producing high integrity systems. The activities of each of the four sub-disciplines are outlined in section 2.3.2, however several key configuration management concepts regarding the constituents of a software system must first be introduced. Aspects of the sub-disciplines and key concepts directly applicable to this thesis will be expanded upon in Chapter 3. Additionally, treatment of the four sub-disciplines and the corresponding key concepts from a wider perspective may be found in [262, 263].

2.3.1 Key Configuration Management Concepts

In any discipline where entities are subject to change there are a number of fundamental concepts that underpin a system for managing these changes. Essentially there must be '*units*' to manage, '*reference points*' on which to compare progress, '*stores*' for completed or semi-completed units, '*records*' of how each unit was manufactured or altered and '*lists*' of which units are held in the store and which make up particular products. The remainder of this section describes these fundamental concepts in relation to software systems.

2.3.1.1 Software Configuration Items & Configurations

The increasing complexity and lengthy time-scales of software development projects have made it virtually mandatory for software to be decomposed into separate functional modules [275]. A **Software Configuration Item** (SCI) is therefore a 'manageable' software entity within a software configuration. Examples of SCIs include source code files, object code files, command files, database files, documentation files, project management charts, test procedures, user manuals etc. [72, 256]. In a more generic sense they may be simply described as any unit of information that is stored and managed [8]. SCIs are the fundamental building blocks of any system. Each SCI must be uniquely identified and should be able to change relatively independently of each other thereby allowing parts of the system to evolve at different rates. Aggregates of SCIs form the **baselines** that represent **configurations** at determined points in time.

2.3.1.2 Versions

Versions arise from changes made to source components or SCIs as a result of the need to correct, adapt or enhance the software. A source version group is the collection of interrelated source components resulting from such changes. Versions may be revisions or variants [376].

2.3.1.2.1 Revisions

A source component, '*b*', is said to be a **revision** of source component, '*a*', if '*b*' has been produced by changing a copy of '*a*'. Revisions are sequential and cumulatively record development history such that later revisions are intended to supersede earlier versions [55, 249]. Perry describes revisions of this type as being 'successive' [311], and Narayanaswamy uses the term 'sequential' or 'vertical' variants [287].

2.3.1.2.2 Variants

Source objects '*a*' and '*b*' are said to be **variants** of each other if '*a*' and '*b*' are indistinguishable under a given abstraction, that is, variants have the same functionality for slightly different situations and as such are intended to be alternative, interchangeable parts and thus are developed in parallel to each other [55, 249, 332]. Perry describes these as 'parallel versions' [311] while Narayanaswamy uses the term 'horizontal variants' or 'alternatives' [287].

2.3.1.2.3 Composed Versions

Configurations also exist as versions and as such must be controlled within the framework of the SCM system [177]. Perry uses the term **composed versions** [311] to describe structures resulting from the combination of several components, e.g. modules from syntactic components, sub-systems from modules [72]. Such structures are represented by a user defined list of unit components and the version to be used.

2.3.1.3 Program Families

A software system is a complex product realised through the interaction of many components. Throughout its life, software evolves to cope with new requirements, enhanced functionality, adaptation to new machines, new operating systems or new customers, or simply to fix bugs. As a result, a software system does not exist as a single 'monolith', but as a family of closely related but distinct systems, each member catering for slightly different requirements [55, 286, 287], and all of which must be rigorously controlled [256].

2.3.1.4 Baselines

Baselines are reference points or plateaux in the development of a system. They should be formally defined at the end of each stage of the lifecycle [65], and be associated with the production of a physical item, that is, either a document or a version of code. Baselines provide an inventory of the current state of a product at one specific point in time and capture both the structure and details of the product. In doing so they serve three connected functions as: a measurable progress point; a basis for subsequent development and control; and a measurement point for assessing quality and integrity of a system [64]. As such they provide an infrastructure for traceability throughout the project ensuring that the product being built conforms to specifications.

Baselines are accepted only after review and audit. Their acceptance is based on the criteria that they meet the requirements contained in previous baselines or, in the case of code, that they are successfully tested against previously defined parameters. Once a baseline has been accepted the document or code should be 'frozen' to form a solid foundation for the next stage of operations. Any subsequent changes to items within the baseline must then be formally effected. The actual baseline itself is never changed, changes are made to copies, and once tested and accepted are recorded as a series of revisions to the configuration. These updates are then collectively assimilated into a new configuration baseline at the next reference point. Indeed, it has been said that the first requirement of change control is to ensure that baseline versions remain inviolate [211].

Major baselines are struck during development at the analysis, design, implementation and operational stages, that is, as system development moves from one stage to the next. In a similar fashion there is a need to identify existing baselines prior to making changes and to establish new baselines at clearly defined points within the maintenance of a software system.

2.3.1.5 Library

Software library systems provide a common and controllable store for all the elements of a software system. They provide a basis around which SCI sharing and change control can be built. They are essentially analogous to book libraries where books can be stored, checked out, read and returned using defined procedures [394]. Software libraries provide features for storing, naming, browsing and borrowing of SCIs as well as providing access security to SCIs. In addition they are often responsible for the automatic or semi automatic production and updating of derivation histories for each SCI in the library [43]. The organisation and implementation of libraries may vary for different system environments and the particular tool being used. For example, some libraries may deal specifically with source code files, and others with object files; some may hold files or functions specific to a particular function or set of functions; and others might be organised to reflect the different states of baselines or SCIs [236, 333, 356].

2.3.1.6 Derivation

The purpose of the derivation or history of a module or program is to record precisely and accurately all the information tracing the evolution of a SCI. Each SCI has a derivation and each derivation references other SCIs and thereby other derivations rather like a family tree. Derivations should identify the tool, the options, the author and the reason for the change [22]. The reasons for derivations are twofold, firstly as an error detection mechanism and secondly to enable previous releases of the software to be reproduced.

Derivations in source files and documents may take the form of a change log, created at the top of each file [372, 373] or may be held as separate files [45], to record why the new version was created and why subsequent changes were made. It is essential that any tool or input listed in a derivation is not deleted or altered.

2.3.1.7 Master Configuration Index.

The Master Configuration Index (MCI) is a list that uniquely identifies all configuration items and from which it is possible at any time to identify the current configuration and the documentation that represents and describes it [57].

2.3.2 Software Configuration Management Activities

The previous sub-sections have described the key physical elements that make up a software product and/or system for controlling the development of that product. However, for successful software product development there also needs to be emphasis placed on control of the process. This control can be realised through the sub-disciplines of software configuration management.

2.3.2.1 Software Configuration Identification

Proper control of a system cannot be effected unless the baselines of the system, the constituent components (SCIs) and any changes to these components are specified and uniquely identified [65]. Nor can software builds and rebuilds occur unless all the constituent parts of the system and where they are located is accurately known. The definition, labelling and representation of changes to software components and baselines are therefore the functions of software configuration identification.

As mentioned in section 2.3.1.1, each baseline is a collection of software configuration items (SCIs) which evolve at different rates. As the system evolves through its lifecycle, SCIs will evolve in parallel, and identification where possible should reflect the name and SCI type, e.g. by the use of suffixes for design documents, requirements documents etc. Any versions created must also be distinguished via naming conventions and registration details. Additionally the baselines themselves should have two labels, the first to identify position in the lifecycle, and the second to represent update level.

Bersoff *et al* liken a system baseline to a 'snapshot of the aggregate of system components as they exist at a given point in time', and updates to the baseline as 'frames in a movie strip of the system lifecycle' such that they collectively represent the evolution of the software during each of its lifecycle stages. Correspondingly, the role of software configuration identification is to provide labels for the contents of these snapshots and the movie strip [42].

2.3.2.2 Software Configuration Control

Due to the dynamic nature of software there will always be the need to incorporate new features, make corrections, and improve code efficiency. Once a baseline has been established and its corresponding SCIs and configurations are 'frozen', any further changes made must be formally controlled and identified thereby allowing all relevant factors to, and effects of, a change to be considered.

The role of software configuration control is therefore to manage changes introduced into the software product, by providing the administrative mechanism for precipitating, evaluating and approving or disproving all change proposals [80]. This is essentially a three component process:

- *Documentation*: to formally identify and define the change, why it is required, expected impact on the rest of the system etc..
- *Organisational body*: to evaluate and approve or reject the change.
- *Control procedures*: to act as guidelines for implementing the changes according to company standards etc. [256].

Software configuration control is concerned with both document and code change management and encompasses three levels of regulation: change control, version control and configuration control. The aim of SCM systems is to control the revisions and variants resulting from changes made to the components of system configurations. The degree to which this is attempted and achieved depends somewhat on the method and tool adopted. Version control and management of the system evolution process have been the subject of much debate and research. However, it is generally agreed, that control is required at two levels, at the level of revisions of individual components (i.e. intra-version group control); and at the level of managing configurations (i.e. inter-version group control). Winkler [399, 400] discusses these in terms of managing program-variations-in-the-small (i.e. changes applied to individual 'building blocks' or components) and program-variations-in-the-large (i.e. building systems out of the program building blocks).

2.3.2.2.1 Change Control

The role of the change control process is to ensure that changes to individual components are made and incorporated in a correct and ordered fashion. Changes to an item are generally instigated in response to requests originating from the user. These requests are documented as change requests or problem reports [104, 141, 223, 271, 329] and define the changes that need to be made. Changes are most commonly made and controlled via a 'check-out, edit, check-in cycle' [315]. A change history for each item within a system should be maintained, detailing 'what' changes have been made, 'when', 'where', by 'whom' and 'why'. The versions arising from a particular component in a system collectively form a version group, all members of which must be efficiently stored within the system in a manner imposed by the configuration system.

2.3.2.2.2 Version Control

Versions arising from changes to a particular component collectively constitute a version group or family [343]. As versions come into existence the relationships between them form a directed tree or acyclic graph structure. In the earliest stage version groups will generally consist entirely of revisions and as such will be represented by a single main branch or 'trunk'. As the version group develops, side branches representing the variants begin to appear. There are two distinct types of branches, namely 'alternatives' and 'elaborations' [147]. Alternative branches arise from modifications for reasons of parallel development (alternative design ideas or implementation details) and must generally be maintained in parallel to the original system. Elaborations arise from modifications for reasons of temporary fixes, distributed development and conflicting developments, and are generally intended for eventual incorporation into the final product, after which development will continue along the main branch. Whatever the reason for the modifications and the type of branch effected the addition of versions adds to the complexity of the overall program family structure. An alternative approach is being considered as part of the VOODOO project whereby variants and revisions of a whole product are managed rather than variants and revisions of individual components [331, 332].

Other aspects of version control include the process whereby the elaborations are merged into the main branch [330] and the space efficient storage and reconstruction of different versions of a software component or configuration on demand. These two issues are outside the scope of this thesis, but are addressed in [66, 262, 263, 330].

2.3.2.2.3 Configuration Control

A configuration is a collection of components that make up a system. As the individual components change and evolve to form version groups, it follows that the configuration of which they form a part must also change and evolve to form composed versions and program families. Mechanisms are therefore required to control the selection of components that constitute a specific configuration at a given instance in time, and the evolution of the system configuration descriptions over time. Both of these areas are the subject of much active research and the reconstruction of these system configuration descriptions for a system at a particular instance in time is one of the main concerns of this thesis. The process of configuration control is also often referred to as system synthesis or program building and may be defined as the management of complete versions and the interrelationships among the components. Approaches to this activity will be discussed in more detail in Chapter 3.

2.3.2.3 Software Configuration Status Accounting

The function of software configuration status accounting is to provide the mechanism and tools for recording and reporting the identities and descriptions of all the SCIs in the system, together with records of the status of proposed changes and the implementation state of approved changes [386]. That is, status accounting provides an administrative history of the way in which the system has evolved. Typically reporting includes:

- Descriptive information about each SCI & corresponding baselines.
- The dates when baselines and changes from the baselines were established.
- The date each item was brought under configuration control.
- Change proposal status.
- Descriptive information about each change proposal.
- The current issue or change status of each SCI.
- The status of proposed changes to SCIs.
- Descriptive information about each change.
- Status of technical and administrative documentation associated with a baseline or update.
- Deficiencies identified by a configuration audit.

Software configuration status accounting therefore enables the state of a system to be reported to a manager or customer at any stage in the evolution of the product. Its records also provide the means by which the master configuration index relating to any baseline can be reconstructed, and they act as a recording mechanism for configuration auditing [256].

2.3.2.4 Software Configuration Audit

Software configuration auditing through a series of reviews and audits is essentially a 'find and report' process. The sub-discipline assesses the technical and administrative integrity of a product, by establishing that change control procedures have been adhered to; that changes have been made in a controlled manner; that no inconsistencies exist; and that the actual content of the baseline is the same as the planned content. In doing so software configuration auditing makes the current status of the software system visible to management.

Software configuration auditing therefore can be considered to serve two purposes: configuration verification and configuration validation. Configuration verification ensures that the specification for each SCI in one baseline or update is achieved in the subsequent baseline or update, whilst configuration validation ensures that the SCIs serve their correct purpose, that is, they meet the customer requirements. Both of these functions are achieved by inspection of:

- The project history (from the accounting process) to identify changes made to the system.
- System baselines to identify relationships between SCIs and correlations between baselines.

The successful completion of these two audit stages is the mechanism for establishing new baselines.

2.3.3 Software Configuration Management Systems

As computer systems increase in size, paper based SCM procedures can no longer be considered adequate to control changes to the components and configurations within these systems reliably. Manual paper based systems are slow, restrictive, error prone and generate a substantial overhead for large computer based projects [256]. Additionally, for large computer systems there are also the problems associated with maintaining consistency, concurrency and security of code when teams of programmers are working on the same project. However, as stated in section 2.3, SCM has the potential for far more automation than hardware configuration management and hence offers great opportunities for SCM-based CASE tool exploitation.

A number of SCM tools are emerging from commercial companies and academic establishments to address the issues of control during the development and maintenance of large software systems. These tools focus on automating areas such as configuration identification, change control, version control and system building. These tools range from simple coincidental utilities available on the host system, through dedicated configuration management systems to, at the most complex level, integrated project support environments. Additionally, there is now an increasing number of languages being developed for the description of software configuration systems, these will be described in detail in Chapter 3.

2.3.3.1 Coincidental Utilities

Coincidental utilities are tools that are intrinsically available on the host system environment. They are not specifically provided for SCM purposes [208] but can be used to provide information regarding software system configurations and hence they can assist the program comprehension process. They must however be used within an established framework; often lack a user-friendly interface and they generally involve a substantial degree of effort if the results are to be usefully applied. Examples of such tools include:

- ***awk***: a pattern scanning and processing language designed to make many common information retrieval and text manipulation tasks easier to state and perform. *Awk* scans a set of input lines in order, searching for lines that match any of a set of patterns specified by the user [5, 208].

- ***cflow***: generates a C flow graph. It analyses a collection of C, *yacc*, *lex*, assembler, and object files and attempts to build a graph charting the external references.

- ***cpp***: the C language pre-processor. Using the '-M' option it will generate a list of *makefile* dependencies and write them to the standard output. This list indicates that the object file which would be generated from the input file depends on the input file as well as the include files referenced.

- ***grep***: searches input files for lines matching a pattern.

- ***lex***: a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in the C language which recognises regular expressions. The regular expressions are specified by the user in the source specifications given to *lex*. The *lex* written code recognises these expressions in an input stream and partitions the input stream into strings matching the expressions.

- ***lorder***: finds ordering relations for an object library. If given one or more archive files, it lists pairs of object file names such that the first file of the pair refers to external identifiers defined in the second.

- ***yacc***: a utility with more powerful pattern matching capability than *awk*. *Yacc* (Yet Another Compiler Compiler) uses a variant of context free parsing to automatically construct parsers for many programming languages. It can be used in combination with *NewYacc* to analyse and transform programs at the source level rather than at the level of the compiled object code [208].

2.3.3.2 Function tools

Rochkind's *Source Code Control System* (SCCS) [335], and Feldman's system build tool, *Make* [153, 146, 364], which were developed during the mid to late 70s may be considered as the pioneers in the field. The last decade has seen an increase in such tools, such as *RCS* (Revision Control System) [373], and *CMS* (Code Management System) [394] for source code control and storage, and *MMS* (Module Management System) [394], *Build* [133] and several other enhanced *Make* tools [21, 55, 93, 145, 152, 197], for automating aspects of the build process. Indeed *Make* and *SCCS* are still in widespread use today.

SCCS, *CMS*, *RCS* and its distributed counterpart, *Distributed RCS* (DRCS) [297] are typical of simple change and version tools. Their function is to manage and control the changes made to individual SCIs. These tools store versions of text in a compact form (via deltas) and offer basic levels of access control. Products known as program librarians, such as *LIBRARIAN* and *PANVALET* also address the issues of change control. These tools provide central library management for the control, storage, recovery, reconstruction and audit of information [211].

Whilst tools such as *SCCS* control changes made to individual components of a system they have no concept of configuration control which deals with the interrelated collections, or configurations, of items. For example, a configuration might encompass design documentation, object code, source code, test data and JCL (Job Control Language). It is the primary functions of *Make* and other such system build tools, to track the relationships existing between parts of a system, to ensure consistency of the constituent parts of a system and to ensure that the end product is built correctly. Tool combination such as *SCCS* with *Make*, and *CMS* with *MMS* go some way to resolving the problem of managing complete versioned systems. However, there is a lack of integration between these tool combinations because *Make* and *MMS* are built on top of *SCCS* and *CMS* respectively, rather than being seamlessly integrated with them. For example, it is not uncommon for ad hoc solutions to the implementation of the SCM process to be created using tools bound together with 'home grown scripts' [237]. The *Shape* toolkit [254, 253] offers some improvement by providing an extended and more generalised interface to the version control system.

2.3.3.3 Software configuration management systems & integrated project support environments

Comprehensive configuration management systems such as *CCC* [356, 357], *Lifespan* [314, 378], *Endevor* [238], *ClearCase* [237] and its distributed enhancement *MultiSite* [12] offer in a single product all aspects of SCM, i.e. change and version control, configuration control, tracking of the status and change to files, increased security etc. In this way SCM systems offer the advantages of seamless integration between the SCM activities and enable increased emphasis to be placed on the more management-oriented activities of status accounting and audit than is generally possible with single function tools. Such tools may be said to represent the first generation in the evolution of configuration management tools [256].

At yet another level of functionality, the recent interest in integrated project support environments (ipses) has seen an increase in the number of project environments with built in SCM support. Examples include: *DSEE* (Domain Software Engineering Environment) [235, 236], *Cedar* [369, 366], *Adele* [135, 137], its distributed counterpart *Mistral* [158] and *Gandalf* [178, 295]. Ipses manage the complete product lifecycle and offer project management support facilities that go beyond those of configuration management [256]. As such, these second-generation tools offer a cohesive and integrated set of tools to support the design and development of software systems throughout the entire lifecycle [341]. Feiler [143] discusses the effect of ipses on systems development in relation to product, process, resource and environment management. Madar [250] designs, implements and evaluates a set of ipse based tools for configuration management. Additionally third generation tools are now evolving which are characterised by their customisability and ability to encapsulate semantic knowledge of the components. This knowledge when coupled with a database query language allows a system build to be instantiated, based on component selection via the component attribute set [256]. A more in-depth discussion on the development of computer-based SCM tools may be found in [14, 79, 263, 383].

2.3.3.4 Discussion of Current Tool Types and Systems

Each category of tool has advantages and disadvantages, these are most readily seen as a trade off between comprehensibility and flexibility. Tools such as *SCCS* used in conjunction with *Make* provide the basic features of configuration management, but offer only limited support for complex tasks that require a broad information base and strict security access. Conversely, highly integrated or specialised systems such as *Gandalf* or *Cedar*, provide greater consistency enforcement and functionality. However, these tools may also result in inflexibility, for example, by requiring that a programmer uses certain tools or by being language dependent. They tend to be more expensive and aimed at large scale projects. This lack of flexibility and being tied down to particular tools is one of the major concerns of users of ipse based systems. With this in mind the *Genos* [140] ipse enables other tools to be used within its environment. Additionally the configuration management part of this ipse is available as a separate product.

Developers of the configuration management systems *Changeman* [395] and *CCC* also acknowledge the importance of users wishing to tailor tools for their own use. *Changeman* can be customised through the Oracle reporting and development tools, and *CCC* has its own macro type language for customisation. The kernel approach of *EPOS* [101] attempts to provide a flexible infrastructure such that any method or tools for software development may be accommodated. Similarly, the *Odin* program development environment [90, 94] addresses the issues of extensibility through its specification language which can be used to integrate new tools into the existing environment. The *Los Alamos Hybrid Environment* [110] has developed an interface that combines features of the VMS operating system with the *CCC* configuration management tool in an attempt to exploit the full power of both the development and configuration management environments. Tailorable environments such as these are aiming to support the diverse range of languages, team sizes, development and delivery platforms, quality and localisation requirements exhibited by today's software teams [72, 323, 381]. Indeed the need for an open configuration management system becomes especially critical when dealing with distributed and heterogeneous systems [249].

2.4 Large Legacy Systems

Whilst software configuration management has been successfully used to control systems under development and subsequently throughout their maintenance, for many existing or legacy software systems this is not the case. Legacy software may be defined as application software created in a previous traversal of a software lifecycle [6]. With these systems, maintenance personnel have to accept systems whose structure may have degraded significantly over many years of fire fighting problems with patched code. The code has also generally grown in size and complexity over the years in which it has been maintained [32, 322]. In these legacy systems the problems of identification and program comprehension are particularly acute in comparison to changes associated with 'green-field' development or those systems produced under controlled conditions. This is due in part to the system degradation but is greatly compounded by several time-related factors that act individually or in combination to precipitate a '*loss of control*' of the system. These factors include:

- ***Exponentially increased component and relationship complexity over time***: as a project increases in size, its complexity increases in a manner that is based on more than just lines of code. Large software projects are invariably complex both in terms of the number and diversity of components, and the relationships existing between them [406].

- ***Systems version explosion***: as a system evolves, new versions of components arise for reasons of adaptation to new hardware or operating systems, tailoring for individual customers, enhanced functionality, or to correct bugs. It can be argued therefore that software does not exist as a single monolith but as families of systems each member catering for slightly different requirements [287, 305].

- ***Lack or loss of useful documentation***: as the system ages, there is a tendency for the documentation to fall out of step with the code changes. This means that even if documentation is available it is highly probable that it is incorrect or misleading [52, 53, 306].

- ***Personnel changes***: in all software projects there is the inevitable turnover of staff. For systems with a long operational lifetime it is highly likely that many of the original development personnel will have left taking with them much of the knowledge about the system. There also appears to be social and technical difficulties associated with recruiting younger staff to the maintenance discipline [234].

- ***Monolithic nature***: the system may have been developed at the time of early programming such that it is inadequately modularised and hence very difficult to decompose the system to a set of smaller more manageable units.

- ***Shifting technology***: as the field of computer science advances new hardware, languages, techniques and environments are developed [Bennett95]. With 'old' or 'geriatric' legacy systems it can be difficult to find maintainers experienced in these earlier languages or environments [4].

- ***Maintenance induced errors***: as system complexity increases so does the probability of new errors being introduced as the system is maintained [34, 96]. This results in 'ripple effect' induced system degradation. For example, it has been postulated, that in connection with the Year 2000 problem, for every 10 million lines of code requiring updating 1,400 defects will be introduced [259].

- ***Heterogeneous techniques***: as systems are maintained over a lengthy period of time, consistency of approach and style to designing and implementing changes become apparent unless definite guidelines have been laid down [4].

Prior to any change being made there is a need to identify and understand the component parts of the system configuration of the system, the dependencies existing between these component parts and the likely impact of any proposed change. If systems have become degraded there may be a need to repair parts of the system or to recapture as much information as possible about the system before these activities can take place. If this is the case, there is a need to recover information about a system which can be done at varying levels of detail and abstraction in order that legacy systems can be better maintained in the future. This activity often takes the generic term reverse engineering, but this umbrella term actually covers a number of different activities which are described in section 2.5

2.5 Reverse Engineering

The concept of reclaiming information from existing or legacy software systems is not novel. Indeed, many legacy systems have become so complex and unmaintainable that full or partial reverse or re-engineering engineering is the only available option if maintenance and management of the system is to be continued at an acceptable level. This is achieved through the potentially stabilising effect on the system of the reverse engineering process which can be used to cope with complexity, recover lost information, detect side effects of changes, synthesise higher or alternate abstractions and facilitate reuse [84].

Over the past decade numerous levels and approaches to the 'reversal' of the software development process have been proposed leading to a classification hierarchy concerning the reclamation of system information. Whilst there is still some argument over the exact definition and nomenclature of these processes [301], they may be broadly divided into the categories of redocumentation, restructuring, reverse engineering, re-engineering [84, 306], inverse engineering [283] and domain analysis [154, 179, 318] according to the degree of design recovery/ restructuring involved. However, reverse engineering, defined as [84]:

> *"the process of analysing a subject system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction"*

is often used as the umbrella term to encompasses the entire activity classification hierarchy. More specific explanations of the terms are given below:

- ***Redocumentation:*** is the simplest form of reverse engineering. Essentially it involves scrutiny of the program code in order to create an alternative, easy to visualise, representation of the program structure to assist the human understanding process.

- ***Restructuring:*** involves transforming the representation of the system from one form to another at the same relative abstraction level, hence it can take place at the levels of implementation, design or requirements. Although it changes the appearance of the system it does not alter the functionality or semantics in any way.

- ***Reverse engineering:*** is a process that combines an element of redocumentation with that of design recovery such that it enables a system to be represented at a higher level of abstraction [51]. It achieves this through the combination of system reclaimed information with that of domain knowledge, external information and fuzzy reasoning or deduction about a system.

- ***Inverse engineering:*** is the term coined to describe the complete reverse engineering of a system from code to specification through the use of formal transformations [390].

- ***Re-engineering:*** involves an element of reverse engineering followed by a period of forward (traditional) engineering during which additional functionality may be added to the original system to correct, adapt or enhance it.

- ***Domain analysis:*** may also be considered in this section since it involves identifying, understanding organising and representing the application context and working practices within which the system has been implemented [117, 234, 379]. Only through this understanding can the 'in-situ' impact of maintenance changes be considered.

2.6 The Interrelationships between Software Maintenance, Software Configuration Management & Reverse Engineering

The disciplines of software maintenance, software configuration management and reverse engineering discussed within this chapter do not operate in isolation, rather they are interrelated within the context of the software development and maintenance lifecycles. The profile of these interrelationships is largely determined by the developmental history of the software system. Additionally for systems that have become out of control through ad-hoc development or maintenance there is a requirement for an additional process to bridge the gap. This process, ISCM, is the subject of this thesis. These interrelationships are summarised in Figure C2-1 and are discussed in sections 2.6.1 and 2.6.2.

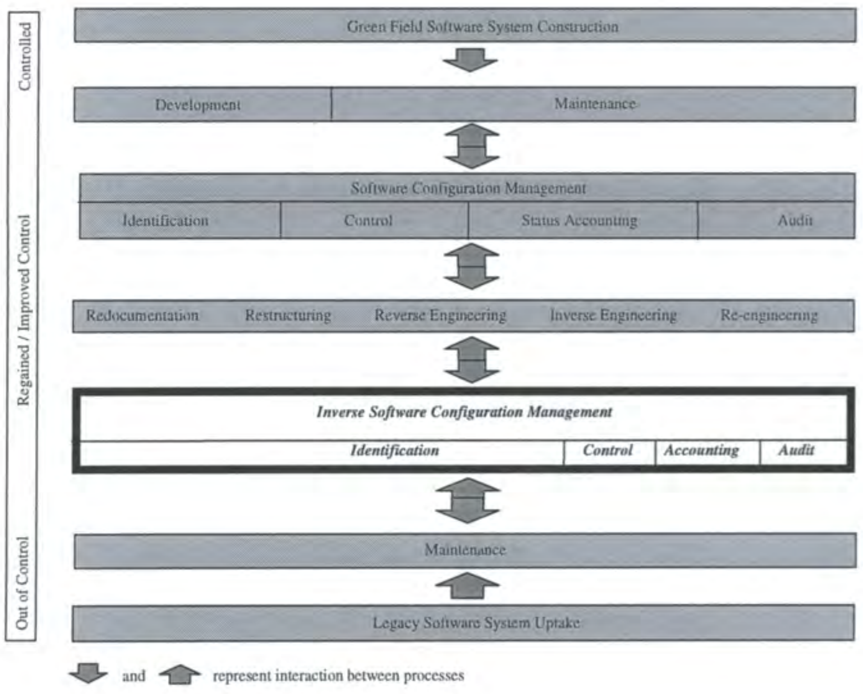


Figure C2-1 ISCM within the global software development and maintenance framework

2.6.1 New Software Systems

In an ideal project scenario, a 'green field' software system will be developed using software configuration management principles from the outset, following a controlled development process, which is subsequently continued throughout the maintenance activities. Effort characteristics of development : maintenance over the entire lifecycle, will tend to exhibit a 33% : 67% ratio. This is due to the continued need to maintain a successful system to enable it to evolve and remain economically viable for as long duration as possible. Even in this ideal situation, an element of program comprehension will be still be required each time a change to the system is to be effected. However for a system developed and documented throughout its lifecycle with maintenance in mind, the time spent on program comprehension will be minimised and the probability of adding errors to the system through maintenance significantly reduced.

For such systems developed and maintained under software configuration management (SCM) principles from the outset, the proportions of effort expended on the SCM sub-disciplines of identification, control, status accounting and audit should be roughly equal. The resultant effect of continual employment of SCM techniques will be a system that remains 'under control' thereby reducing comprehension time and easing development and maintenance. From this it can be seen that software configuration management and software maintenance have a very strong bond connecting them. Software maintenance drives the software configuration process, in so much as, without maintenance there would not be the need to recreate new system configurations nor to compose a program family consisting of a series of viable configurations assembled from the members of component version groups. Conversely, in the absence of software configuration management, the software maintenance process very rapidly gets out of hand and the software products spiral towards uncontrollable systems.

As a system progresses throughout its life, even with controlled development and maintenance there may come a point when the system structure starts to degrade and an element of redocumentation, restructuring, reverse engineering or re-engineering becomes necessary. In the case of controlled systems this may be minimal in terms of the proportion of the system affected and the process itself is facilitated through the provision of a solid foundation on which to base the reverse process. Evolution of the system will then continue as before with all changes to components identified, implemented and recorded in accordance with SCM guidelines and procedures.

2.6.2 Legacy Software Systems

It is a different scenario for legacy software systems. Legacy systems are entirely maintenance driven since their development phase has generally been completed many years earlier and often by a completely separate team. Maintenance therefore accounts for 100% of effort since even the addition of new features will be classed as perfective maintenance rather than new systems development.

If these legacy systems have been developed and maintained under SCM procedures, future maintenance can continue in a similar fashion to that described in the previous section for 'green field' systems. However, many legacy systems have become 'out of control' with the result that the system configuration has, at best become undocumented, and at worst has become corrupted or obscured with many missing or redundant SCIs. The level of program comprehension required before any changes can be safely made to these legacy systems is therefore often very high and hence maintenance of such systems very expensive.

To enable these systems to be maintained safely and cost effectively, there is a requirement to re-establish a record of what the system configuration is; to ensure that each component of a system configuration is uniquely and correctly identified; and to assess the degree of dependency of each component on or to the other components of the configuration. If the configuration can be regained program comprehension of a legacy system can be effected much more readily and control continued in a process akin to that of systems developed from the outset using SCM principles and practices.

It is in this area of reclaiming the underlying system configuration that key advances can be made in the fields of software maintenance and software configuration management. There is an identified need to create a process that will enable the reclamation and reconstitution of the time-and-complexity-corrupted system configuration. Inverse Software Configuration Management (ISCM) is the process being defined in this thesis as the mechanism for enabling this to occur. Whilst recognising the importance of incorporating the four sub-disciplines of software configuration management within the ISCM process, the relative proportion of effort expended on the identification sub-discipline is much higher in the ISCM process than it is in the traditional SCM process. This is due to the significantly increased proportion of comprehension work involved in re-identifying the configuration of a system corrupted and degraded over time. However, once identification is achieved, the other sub-disciplines of control, status accounting and audit can proceed in a manner very similar to that of the traditional SCM process.

Employment of the ISCM process may highlight, that part or all of the system is in need of 'repair' through redocumentation, restructuring, re-engineering or reverse engineering activities, the early stages of which may, in themselves, contribute to the information collection process. The ISCM process and the newly identified system configuration subsequently provide a solid foundation for the later stages of the reverse processes to proceed. Indeed, without the ISCM process, substantial re-engineering is very difficult if not impossible due to the indeterminate starting point of a corrupted configuration.

Once control has been regained through the ISCM process, evolution of the system can continue safely whether via forward or reverse engineering activities.

2.7 Software Maintenance Documentation

Whatever the type of maintenance work carried out it can be argued that the process is only really useful if the knowledge gained during comprehension can be documented in such a way that it can assist future maintainers. It has been shown [46] that, in order to maximise understanding, this documentation should be a mixture of information types encompassing both high level domain knowledge and low level program detail. Although development would ideally realise 100% documentation of the software creation process and resultant system, it has been widely stated that 80% of maintenance time is attributable to only 20% of the code. Hence, it may follow that retrospective detailed documentation need only be conducted on the parts of the system requiring maintenance. Additionally, many of the problems of system comprehension are compounded by the lack of reliable, up-to-date and maintenance-oriented documentation, due to the overhead of producing or updating documentation in step with maintenance activities. These two factors lead to the concept of incremental documentation that is implemented in a number of systems [76, 148, 149] as well as by the ISCM process [264, 265].

Traditionally, much of the comprehension process during maintenance has centred on scrutiny of the low level program code in an attempt to work out the 'what if' effects of changing a particular module on the other parts of the system. This process is termed *impact* or *dependency analysis* and if not conducted properly may result in many more errors being introduced into the system than were removed by the change sometimes with catastrophic consequences [187]. It is therefore particularly important during maintenance that this impact information can be obtained as rapidly and as accurately as possible. Additionally, current-thinking postulates that this knowledge, in order to be really effective, should be coupled with information types at a higher level of abstraction and as a mix of informal, semi-formal and formal documentation types [225]. This is because comprehension difficulties may also be attributed to the loss of valuable domain knowledge, historical data and co-ordinating information [225, 234], that occurs when an application crosses the 'time-space divide' [267] between development and maintenance. This information is however fundamental in ensuring that all personnel are working to a common definition of what system they are building, what it should do, how it should be organised and how it should be integrated with previous systems already in place. Indeed, mapping of the problem domain knowledge with the program domain functions and structure is considered essential if full comprehension is to be achieved [368]. There is thus a need to study and understand the linkages at a level of abstraction higher than the source code and to be able to document this knowledge in an informative manner.

The process of software maintenance and program comprehension in particular has only recently been given the attention it deserves, with theory, tools and techniques now being published alongside those for the development process. Many of the methods to aid program understanding, concentrate on identifying key structural features or 'beacons' within a program [60] or on representing the structure of the code in the form of flow charts and graphs [293, 46]. Other methods concentrate on representing documentation such that the programmer gains a 'wide picture' of the system. These methods include the 'book paradigm' which is an arrangement of source code into an on-line book [298], and the use of hypertext to correlate program understanding with the human semantic network [52] or to link program dependency information with the syntactic structure of the code [302]. The ISCM method takes the approach a stage further by concentrating on linking the program dependency information with documentation at higher levels of abstraction [264].

Whilst the use of hypertext has become common place for documentation in areas such as application help systems and is becoming increasingly popular as a means of documenting software development and maintenance [52, 159, 160, 202, 319], the use of multimedia and hypermedia in the software maintenance process has not yet been fully exploited. Although program comprehension during the maintenance process necessitates a strong focus on the study of the program code, it appears that it may be useful to incorporate other information sources collated during the maintenance process. As mentioned previously, this information may be of an informal (personal, peer-oriented, unstructured communication), semi-formal (personal, fairly structured, review type communication) or formal (impersonal, written, structured communication) nature, the ratio of types depending on the application being developed [225]. There is thus enormous potential for multimedia technology to be used to capture this missing domain and co-ordinating knowledge in a unique way that exploits video, audio and animation, which when added to the more common textual and graphical representation will provide a more productive and '*value added*' program comprehension/maintenance environment.

2.8 Role of Inverse Software Configuration Management

There is no doubting the importance associated with today's role of Software Configuration Management (SCM). SCM is increasingly becoming mandatory in software engineering, is required by a number of standards and is considered as the major tool for controlling software production [138, 139]. What is now required in the maturing software engineering discipline is for the principles and techniques of SCM to be made truly applicable to legacy software systems. SCM for developing systems encompasses, in approximately equal proportions, the activities of identification, control, status accounting and audit. These activities also form the basis of the ISCM process, however within ISCM there is a shift in emphasis towards the identification activity, since this forms the critical underpinning of the subsequent control and reporting activities. Indeed, enabling identification of legacy system components and their interrelationships will in itself significantly decrease maintenance costs due to the reduction in comprehension time prior to any maintenance change.

As well as clearly defined activities, successful SCM systems have a number of fundamental physical concepts associated with them, namely:

- Units to manage (components and configurations)
- Reference points (baselines)
- Stores (object base)
- Records (administrative documents)
- Lists (configuration descriptions).

It is therefore considered important that these elements are incorporated within the context of the ISCM process. As this thesis progresses it makes apparent how the ISCM process enables the **components** of a legacy system and its surrounding environment to be identified and descriptions of their corresponding **configurations** reassembled at varying levels of abstraction. This has the effect of enabling the operational **baseline** for a legacy system to be identified and used as a reference point for subsequent control and maintenance of the system. The **configurations descriptions** are documented by means of an Inverse Configuration Description Language (ICDL) developed as part of the ISCM framework. The ICDL essentially lists the components of a configuration, their attributes and their dependencies. In any configuration management system it is important that comprehensive **administrative records** are kept. The ISCM process therefore also encompasses the definition of a series of proformas and forms for incrementally recording the information reclaimed about a legacy system and the status of any maintenance activities being conducted on these components. **Storage**

mechanisms for the ISCM process rules and resultant reclaimed information are also implemented.

ISCM provides the opportunity for an innovative approach to the configuration management of legacy software systems. It also approaches the program comprehension process from a different perspective in that it combines traditional code level program comprehension techniques with technology-enabled video, audio and animation techniques to provide comprehension at a higher level of abstraction. However, it is not the intention of this research to investigate program comprehension *per se*, that is, at the level of program slicing, call graph generation etc. [87, 131, 174, 363]. Rather, the aim is to study the application of SCM procedures and techniques to help enable the program comprehension and subsequent maintenance and control process. Additionally, through the use of hypermedia links between the different information types, an original and enriched documentation interface and environment is created for the recording and reporting of software system configurations which meets the needs of maintainers.

A number of tools are emerging from commercial companies and academic establishments to address the issues of SCM. *Make* and *SCCS*, developed during the mid to late 70s are considered to be the pioneering tools. These tools are still widely in use today and have also served as a basis for newer and more comprehensive configuration management systems. The deserved recognition of SCM as an integral part of the software development and maintenance process is evident from its inclusion in the majority of ipses that are available today. However, whilst these tools offer the functionalities of SCM many of them lack the flexibility to be tailored explicitly for a customer's environment [323, 383]. They can also be very costly [2]. With these two factors in mind a meta-CASE environment has been developed to assist and semi-automate the ISCM process. Provision of a 'meta' environment enables incorporation of flexibility and cost effectiveness in terms of being able to make use of whatever host resident tools are already within the software system environment.

SCM is not just concerned with the obvious components such as source and object code but also with all associated data and documentation. Examples include requirement documents, design documents, source code, binaries and executables, graphics files, control language files, test results and data, planning documents etc. Whole environments such as compilers, libraries and pre-processors must also be brought under software configuration management control [103]. This is reflected in the definition of SCM given by McCarthy which states that configuration management, covers the management of 'every detail' of a software project through its lifecycle [260]. Indeed SCM may not just be limited to control of the products of the project lifecycle, it may also be extended to include the processes as well as the products occurring during the project [65]. Correspondingly, ISCM may therefore be considered to cover the management of 'every detail' and process activity associated with a software product during its maintenance phase [271].

Thus, although progress has been made in the adoption of SCM principles and techniques for control of new software systems, there still exists a need for a process that addresses systems primarily from a legacy perspective. This problem is addressed within this thesis through the provision of an ISCM framework. The ISCM framework is centred around the definition of a new process which is integrated into the lifecycle via a conceptual process model. The process model is brought into practical usage through the development of a clearly defined method which in turn is driven and semi-automated through the development of a meta-CASE environment. Development of the ISCM approach and its distinctiveness from current SCM and program comprehension methods will be described and discussed within the remaining chapters of this thesis.

2.9 Summary

This chapter has described the fundamental principles of the software maintenance, software configuration management and reverse engineering disciplines. It has also examined how these disciplines exist, not in isolation, but as an interrelated mix of activities occurring in varying proportions throughout the lifecycle of a software system. Detailed study of these inter-relationships has revealed a gap in the current lifecycle approaches to controlling changes to and further development of legacy systems. This has thus laid the foundation for the creation of a new software engineering process, Inverse Software Configuration Management (ISCM). The purpose of ISCM is to bring legacy software systems back under configuration control. The resultant effect of this is that legacy systems may then continue to be maintained safely and effectively whether for corrective, adaptive, perfective or preventive maintenance purposes. Definition and development of the ISCM process is the primary aim of this thesis. The ISCM process and its unique features will be described in Chapter 4. However, Chapter 3 first explores in more depth some of the key issues associated with managing and modelling software system configurations namely maintenance process models, software configuration components and methods for modelling and constructing such configurations. It is on these concepts that the ISCM process is based.

Chapter 3

Managing & Modelling Software Configurations

This is as strange a maze as e'er men trod,
And there is in this business more than nature
Was ever conduct of. Some oracle
Must rectify our knowledge

Shakespeare The Tempest, Act V, Scene I

3.1 Introduction

Software configuration management (SCM) has been successful in reducing costs for systems under development by enabling on-going control of the product. For operational systems, it follows that if control can be regained, thereby emulating a system produced using SCM principles from the outset of development, then more cost-effective maintenance should be achieved. This thesis develops a new process, **Inverse Software Configuration Management (ISCM)**, which may be defined as *the process of bringing existing (operational or legacy) software systems back under configuration control.* The driving force behind the success of the ISCM process is the combinatorial effect arising from the application of traditional SCM principles and practices within a controlled software maintenance process framework. Chapter 2 introduced the fundamental concepts associated with software maintenance and software configuration management. This chapter extends and focuses the review in a number of key areas. Firstly, it outlines the current body of knowledge regarding maintenance process models, highlighting the individual strengths and weaknesses of each model and from this deduces a number of features for incorporation in the ISCM process.

Secondly, the foundation of the ISCM process and its associated method is the reconstitution and allied unambiguous representation of software system configurations or architectures. This chapter therefore also reviews the constituent components of a software configuration (architecture) and outlines a number of approaches for syntactically describing such configurations. In forward engineered systems these architectural descriptions can be used to build or manufacture a viable software system from the possible component types and the respective versions of these types. This chapter therefore also reviews the configuration synthesis process as a means of determining how to reconstruct the viable system configuration description from the chaotic and corrupted *out of control* legacy software during the ISCM process.

Thirdly, to understand the configuration and to model it accurately there is a need to reclaim as much knowledge about the system as is possible. However, as stated in Chapter 2, it is likely that only part of a software system will undergo comprehension during a particular change request [245]. Thus knowledge elicitation about a complete software system architecture is generally progressive over the entire maintenance sub-lifecycle and may be in the order of years. It is therefore essential, if maximum process efficiency is to be achieved, that information recovered during one maintenance change is not subsequently lost prior to the next required change. Hence there is a need to not only identify any information sources pertinent to system understanding, but to also store all reclaimed information within a data- or knowledge base, in such a way that it can be readily accessed and recomposed into a coherent and correct framework at a later date. The chapter therefore also outlines a number of possible structures for the underlying Extensible Systems Information Base (ESIB) created as part of the ISCM process.

3.2 Software Maintenance and the Software Lifecycle

Although the terms 'software process model' and 'software lifecycle' are often used interchangeably, a clear distinction may be made between the meaning of the two terms. A software process model may be defined as a series of activities whilst a software lifecycle puts this series of activities into a particular cyclic order [386]. A complete software lifecycle may therefore be composed of a software development process and a software maintenance process. Indeed, if representation of the vital management aspects associated with the development and maintenance activities are also added, this may be considered to be a '*software engineering lifecycle*'. Allied to this, a model may be considered to be an abstract representation [272] of a process and a method the abstract representation of the defined activities required to physically realise the abstract model. These terms will form the basis of the definitions used within this thesis concerning the ISCM process and its associated model and method.

3.2.1 Traditional Software Process Models and Lifecycles

Software maintenance is now considered to be one of the key disciplines within the software product lifecycle. In recognition of this, awareness has begun to grow regarding the deficiencies of traditional software lifecycle approaches to the software maintenance process [256]. The main area of inadequacy of many current models is the representation of software maintenance as an isolated operation after delivery of the system to the customer [227]. This has had the effect of maintenance often being viewed as an adjunct of secondary importance to the software development process. Consequently, maintenance is often considered to be a single step process rather than an ordered and planned sequence of activities. Indeed, close parallels may be drawn in process terms with the way in which the software within a system was treated as a single entity in the early days of SCM [22]. There is therefore an imperative need to redefine the global software lifecycle and corresponding models to accurately represent the complexities of the software maintenance process.

3.2.1.1 Code-Use-Fix Model

The traditional software lifecycle approach has in itself developed considerably over time [351]. Originally software development occurred via an 'ad hoc' two-phased process of 'write the code' and 'fix or add to it' as required. This 'code-use-fix' model was feasible in the early days of software development when software was created by single programmers for their own use [368]. However, as greater reliance was placed on the software components of products and as product size increased to such an extent that development became a long term, team build activity for divergent customer groups, the need for a more ordered and accountable approach became evident. Consequently, this led to the development of the first in a series of 'waterfall' based lifecycle models [337, 361].

3.2.1.2 Waterfall Model

Whilst the waterfall model adequately describes the main activities of the software lifecycle (requirements analysis, specification, design, implementation, testing, maintenance), it may be considered misrepresentative in terms of the implied proportions of the effort involved at each stage. This is particularly apparent in the ratio of development to maintenance effort in that 80% of the model relates to development activities and only 20% to those of maintenance. However in real terms these figures are reversed with maintenance activities accounting for up to 80% of expenditure over the lifetime of a software system [10, 243]. Hence there is a need to review the way in which maintenance is represented within this model.

Additionally, the waterfall model even with the incorporation of the validation and verification feedback loops, essentially implies a sequential ordering of development activities each occurring once only, followed by a period of maintenance. In essence, all software is considered evolutionary in nature, continually adapting to fit the needs of its environment [239]. It is more representative of the real-world process to view maintenance as an iterative sequence of development processes with a period of program comprehension associated with each one [361].

3.2.1.3 Evolutionary Development

Other models of software development have been proposed which reflect the evolving nature of software products. These models include the evolutionary and incremental prototyping approaches [26, 191]. Both models may be considered closer to the paradigm of maintenance as they involve an iterative cycle of small phased developments that build towards the final product. However, there is still an important distinction between the development and maintenance of software products. Prototyping approaches have goals of a pre-specified product to be built within a finite amount of time with clearly allocated resources, whereas maintenance is usually open ended and much less clearly defined. A risk associated with prototyping models is that the documentation may not be kept concurrent with the development of the product, thereby creating maintenance difficulties from the outset. This is particularly problematic in the evolutionary approach where it may be argued that the product may change quite significantly from one cycle to the next.

3.2.1.4 Spiral Model

Boehm's spiral model [50] goes some way to addressing these problems. The spiral model combines the prototyping approach (or indeed other approaches) with the traditional stages of software development by responding to feedback from earlier activities in order to reduce risk in later ones. Importantly, by putting the option to prototype within the more strictly defined framework of the traditional lifecycle stages, control over documentation and development can be more closely monitored.

3.2.1.5 Automatic Programming and Transform Model

This model has been developed to account for the likely increase in the number of automated software development tools in response to economic pressures and theoretical advances [24, 256]. The aim of this model is to represent the automatic process of specification capture and validation, followed by automatic conversion of the specification into source code. This may have far reaching repercussions on maintenance in the future as it involves a paradigm shift from changes being made to the source code, to changes being made to the specifications. However, whilst the approach would be effectively reversed, the processes would be analogous in terms of the requirement to recompile the system following a change [256].

3.2.1.6 SEI Capability Maturity Model

The Capability Maturity Model (CMM) is a meta software lifecycle model that has developed from a software technology transfer programme at the Software Engineering Institute (SEI) at Carnegie-Mellon University [115, 339]. This project, funded by the United States Department of Defence, was initiated as a study into how to improve the capabilities of the United States software industry. In particular it was aimed at assessing the capabilities of contractors bidding for projects. This model is rapidly growing in stature as a way of improving control of development and as such should have advantageous 'knock-on' effects for facilitating future maintenance particularly when combined with techniques such as Defect Causal Analysis [77]. The model has been exceedingly influential in convincing the software engineering community to take process improvement seriously and has already produced highly encouraging results [167] although not without some cost and effort [194]. The CMM is based on adaptation of the quality principles of Deming, Juran and Crosby [113, 124, 210] to the software process [320]. This high-level model provides a framework for identifying five levels of maturity that lay successive foundations for continuous process improvement. As such the model centres around determining the current state of the production process and providing guidelines for deciding which areas should be improved and the order in which these improvements should be addressed. It is thus a management oriented model ranging from the 'Initial Level' (very poor management) through to the 'Optimising Level' (very high management and quality).

3.2.2 Process Models for Maintenance

Although development models are moving towards a more accurate representation of system characteristics and natural evolution patterns, there is still a need to take into account the maintenance specific characteristics caused by the *'time-space-divide'* [267] between the development and maintenance processes and personnel involved. This is particularly important for legacy systems whose quality and structure have generally degraded over many years of service and preservation, thus necessitating special attention during the maintenance process. Models of the maintenance process are therefore continually being developed in an attempt to more accurately capture and control the maintenance process and depict net maintenance activity.

In order to address adequately the maintenance process, models need to define the activities that occur during maintenance, at a lower level of detail. Various maintenance process models already exist [27, 34, 48, 49, 76, 228, 299] which address the maintenance process from a number of different perspectives (e.g. economic, task-oriented, iterative, reuse, request driven, reverse engineering). Although these models describe the maintenance process in varying levels of detail, they all centre on the evolutionary nature of software. It must be stated however that these models are still in a state of flux as they are not yet so fully developed or as well understood as the models for the development process [368]. The following sections describe a representative sample of such maintenance process models.

3.2.2.1 Quick-fix Model

The most ad-hoc of maintenance models is the inadequate but frequently necessary 'quick-fix' approach. This model is characteristic of the 'fire-fighting' approach which involves adding emergency patches to a piece of code to keep it operational with minimum disruption to the customer base whose business relies on constant availability of the system. However, this model is also representative of 'ripple effect' problems and the increased need for future maintenance. Such a model can only work if it is encompassed within or coupled to another more rigidly defined model [230].

3.2.2.2 Boehm's Model

Boehm has put forward two models of maintenance. His original model is a three phased approach consisting of understanding the existing software, modifying the existing software and revalidating the modified software [48]. Economic models and principles have subsequently been added to form the foundation of his later model [49]. This model views the maintenance process as a closed loop cycle driven by management decisions. Changes are approved on the basis of cost-benefit evaluations. Essentially, changes continue to be made until the point of 'diminishing returns', that is, the stage when a product has reached its maximum usefulness and any additional change is no longer deemed cost effective [368].

3.2.2.3 Osbourne's Model

This model is based on continual iterations of the software lifecycle with the provision for building in maintainability as required. Particular emphasis is placed on ensuring there are adequate management communications, control, verification and feedback within the cycle [299]. The model combines elements of the traditional development approach with explicit review and audit procedures at the end of each lifecycle stage. Although reminiscent of the traditional development approach in that the activities of the model appear successive in nature, the evolutionary nature of software is represented by the repeated iterations of the activities within the cycle during the period of maintenance.

3.2.2.4 Reuse Oriented Model

This model [27] is based on viewing software maintenance as re-use oriented software development. As such it has four main steps: identifying candidate parts of the old system for reuse; understanding these system parts; modifying old parts to satisfy new requirements and integrating modified parts into the new system. This model is likely to gain in popularity as reuse of components, designs and specifications becomes more widespread within the software engineering community. It is not unreasonable to assume that eventually 'plug and play' software development and maintenance will reach analogous levels to those utilised by current hardware systems [290, 344]. Indeed, significant advances are already being made in these areas due to the introduction of Component-Based Software Engineering (CBSE) [292, 120], Commercial-off-the-Shelf (COTS) products [25, 62, 380] and the establishment of standards and component frameworks such as OLE (Object Linking & Embedding) and CORBA (Common Object Request Broker Architecture) [3]. Such models are becoming realistic as the object-oriented software development and maintenance paradigm becomes more widely adopted [173, 242, 276], particularly in areas such as graphical user interface development.

3.2.2.5 Request Driven Model

The request driven model [34] acts in response to requests from customers for changes to the software. It consists of three main processes: request control, change control and release control and is therefore heavily influenced by elements of the configuration management process. The initial request control step concentrates on the help desk collection of proposed change details, cost-benefit assessment of these changes and prioritisation of accepted changes. The change control activity necessitates analysis of the existing code to understand the system and to ensure limited ripple effect, plus design and implementation of the change within a quality control framework. This is followed by release control of the product into the live environment, again with emphasis placed on quality control and audit of the process. Another model with a similar emphasis on the maintenance process is that proposed by Harjani and Queille [183]. This model is again triggered by change requests, although it differs with regard to its more prescriptive approach to the activities which must be carried out to effect the change and re-insert the corrected module into the overall system.

3.2.2.6 CONFORM Model

This model [76] also centres around the configuration management process. The CONFORM (CONfiguration management FORmalisation for Maintenance) method provides guidelines and procedures for a change control framework, again with an emphasis on incorporating quality control into the change control process. This model is based on the waterfall lifecycle and hence has some similarities to Osbourne's [299] model. However, it includes a specific change evaluation phase and tracks the status and visibility of a change through a series of change related documents that are produced as the output of each lifecycle stage. In this manner, the model builds up a maintenance history of the operational system. This history however is at an abstract level as it does not deal with the source code at all.

3.2.2.7 The 7-Level Model

This model concentrates on mapping an organisation's approach to maintenance into layers ranging from high level 'asset' and 'portfolio' management, to 'topic' (maintenance function) management at the lower levels of the model. This enables viewpoints to be isolated and particular aspects of a problem to be addressed [150, 151]. Central to the model is the maintenance unit or team and the emphasis placed on the functions performed by the team. As such, this approach can be abstracted to model teams of widely differing sizes, since it makes no assumptions about the mapping of duties to actual people or groups of people. Again the model is that of a front desk / help line support but differs from other models in its tight linkage to a 'change store', which locates already documented solutions to reported problems or change requests prior to generating new solutions to uniquely reported problems.

3.2.2.8 Reverse Engineering Model

This model has been proposed in response to the draft IEEE standard for software maintenance [199], which failed to address the issue of reverse engineering [228]. The reverse engineering model uses language-specific code-level (metrics, static analysis information etc.) and language-independent code-level (control-flow complexity, entity relationship information, derived object classes, code fragments etc.) to build up design and specification information via the reverse engineering process. This model essentially involves a process of abstraction whereby the program code is translated to a procedural intermediate language and subsequently to the design level. This process results in a reverse engineered description of a system or system fragment, however if changes are to be implemented a degree of re-engineering of the system will also be required using the reconstructed design.

3.2.3 Discussion of Current Maintenance Process Models

The above section has outlined a number of maintenance process models. From the research it is evident that although maintenance process models differ in their degree of process granularity and in their particular emphasis, they all considerably extend the view of maintenance implied by development models (with the exception of the 'quick-fix' model). Other characteristics important for the maintenance process can also be observed within the different models, for example, their aim to enable effective communication; to support cost-effective maintenance; to facilitate a reusable process; to support evolution by serving as repository for modifications and to facilitate effective planning and increased understanding of the system being maintained. There is also considerable emphasis placed on ensuring that quality that might have been lacking in the development process is installed into the system via the maintenance process. This is evident through the explicitly documented review and audit activities within the models, in comparison to their implicit presence within the development models.

One of the most important details to note about the maintenance models is their reliance on the maintainer being able to analyse or understand the proposed change and existing system at the initiation of the maintenance process. However, despite recognition of this need to comprehend the system, none of the current models, with perhaps the exception of the reverse engineering model [228], detail in any way ***how*** to conduct the process of analysis or program understanding within the maintenance model framework. This is essentially a configuration management problem of component and configuration identification plus related dependency or impact analysis to assess the potential ripple effect of implementing the proposed change(s). Allied to this, it is also interesting to note that although configuration management principles are applied in many of the models, these principles are primarily aimed at documenting and describing the software process via the elements of software configuration control, status accounting and audit rather than in ***identifying*** the system configuration. This may be regarded as an omission in the modelling process since in order to understand a system well enough to be able to make 'safe' ripple free changes, the configuration and related components should be explicitly identified and understood. Once, identification has been effected, the other configuration management elements can be applied during all subsequent activities to maintain control and accountability of future maintenance changes.

Another noticeable characteristic of current maintenance models is that although they describe the maintenance process in more detail than the corresponding development models they are still at a surprisingly high level of abstraction in process terms. That is, the models document the high-level process stages but give little indication of the detailed steps required to carry out the documented activities. Whilst this offers flexibility in terms of their applicability to controlling software production over a wide range of systems and environments, it does mean that there are still many areas for activity contradiction thereby allowing inconsistencies to manifest themselves within the maintenance process. Whilst a high-level of process granularity may be acceptable for systems under conditions of controlled development a more rigorous and prescriptive low-level process model is required to enable 'out of control' systems to be brought back under configuration control. It is in this area, and in particular the definition of a model aimed primarily at maintaining legacy software systems that this thesis and the Inverse Software Configuration Management (ISCM) process model is aimed and developed within Chapter 4.

3.3 Software System Architectures

Techniques for the modelling of software systems have advanced considerably over the past decade. This may be mainly attributed to the now widespread acceptance of the importance of the software within a system, and the need to understand its complexity if the system is to meet and uphold quality, reliability and safety requirements. However, particularly for large systems, identification of the design of the overall software architecture still emerges as a central problem [162, 207]. This may be attributed to several causes prevalent in modern systems development each of which impacts considerably on the level of complexity of software systems:

- ***Internal complexity of software***: software complexity at a program (individual SCI) level has increased due to incorporation of more sophisticated algorithms for the processing of data or controlling of processes.

- ***Number of components***: software can no longer exist as a single monolithic component. The size of software systems today dictates for reasons of technical viability, performance and control that software must be composed of many smaller units or modules (SCIs) which combine to form a complete system through the definition of their module interfaces. Whilst this potentially makes reuse of individual components and maintenance of complete systems easier, it does mean that the relationships between the individual component parts must be identified and maintained if 'ripple effect' problems are to be avoided and systems maintained efficiently and safely. These relationships between components may be very complex in some systems.

- ***Heterogeneous mix of components types***: not only has the number of components increased but so too has the diversity of the component types [203, 292]. This is particularly evident in relation to the widespread adoption of multimedia technology into almost every aspect of society and corresponding computer systems. This much more heterogeneous mix of application component types adds another level of complexity to the modelling of software systems since there is the need to maintain referential integrity of the linkages set up between widely different component types.

- ***Linkage with immediate environment*:** additionally, there is now increased reliance on the integration of application systems with environmental components if a fully working system is to be produced and maintained [306]. This has always been evident in terms of recording information regarding the relationships with tools such as compiler and linker versions and the parameter settings required to create a given application system. However, the complexity of today's systems and the emphasis on reuse of components means that the need to carefully preserve relationships has been extended to include not only application libraries and versions of tools, but also specific system libraries and possible third party library components. Thus there is a need to identify the different component types not only of the application itself but also of its immediate system environment.

- ***Linkage with secondary environment*:** as well as increased reliance on the direct relationships of applications with system libraries, in many systems it is now necessary to consider their linkage with interface layers such as those of Windows 3.1, Windows 95, Windows NT, Motif, X-Windows, Microsoft Foundation Classes, and Visual Basic etc. Additionally, the software may 'sit' upon third party software such as databases, or in alternative scenarios may be evident as firmware embedded within hardware components [65]. There is thus now the need to consider these extra levels of complexity when modelling software system architectures.

- ***Wider application base*:** software has also expanded into a much wider range of business and industrial domains [62, 166] each requiring specific treatment and emphasis on the relative importance of different aspects of the modelling process. Particular mention should be made of multimedia, world wide web [38, 56], embedded [171], distributed [100, 345], real-time and safety-critical [345] systems . It is hence important to understand and be able to document underpinning domain knowledge about each of the different system types.

- ***Change in architecture*:** rapid changes in hardware and software architectures are transforming the nature of application software systems [328]. For example, the rapid growth of client-server architectures is having a major impact on software design in that the client part of these architectures runs the user interface and some of the business logic whilst the servers run a database such as Oracle, Sybase or Ingress [292, 328]. Both the need to interface with 3rd party applications or components and the resultant distributed nature of the software are having major impacts on the complexity of systems [292]. These types of changes are being aided by the advent of Rapid Application Development (RAD) domains, middleware services [39] and languages such as Power Builder, Visual Basic, Delphi, Oracle CASE and Visual C++.

- ***Fragmentation of upgrade paths***: the rapid changes in architecture are also having a significant effect on systems development and maintenance in that upwards of 50% of the market cannot cost effectively integrate new technologies into their systems. In particular this impacts on a company's upgrade strategy in order to take into account changes in the underlying hardware or operating systems [16, 18], and the need to incorporate legacy systems from several preceding generations [203].

- ***Redefinition of application boundaries***: there is an interesting redefinition emerging between what is considered off-the-shelf-packages and in-house development [328]. For example Microsoft is encouraging application developers to build custom solutions by configuring and 'gluing' together standard Microsoft Office components using Visual Basic and OLE. This is leading to a change in the role of the developer / maintainer from that of 'component builder' to one of 'solution provider'. This has implications both in terms of how system architectures are modelled and in terms of the developer now needing to understand more fully the business domain and how to assemble and build a viable system rather than just being technically able to program [328].

- ***Extension of geographic boundaries***: with the large multi-national organisations it is now not uncommon for development and subsequently maintenance to occur at geographically separate sites requiring co-ordination across different platforms and time zones [237].

- ***Linkage with supporting information***: the need to understand a system fully also puts increased emphasis on being able to link with documentation and other specific components such as requirements and design specifications much of which are now stored on-line and which should be incorporated into the overall system configuration for a product [360]. This is particularly relevant in web and other hypertext based material where links are set up between and within different documents or media types [56].

The increased scope of possible system architectures caused by the above factors makes modelling of software systems a much more challenging task than a decade ago when software systems were of a much more uniform nature and standard defined pattern of development [348]. The enormity of the task is further evident when we consider that even a decade ago software systems were being described as the:

> "*most intricate and complex of men's handiworks requiring the best use of proven engineering management methods*" [59].

To understand the increased complexity there is a resultant increase in the need for comprehensive documentation of the reclaimed configurations. There is also the requirement to persistently store the information reclaimed about a system if the effort expended on program comprehension is to become progressively reduced as more knowledge becomes known about a system. For this reason this research has defined a number of **component groups** which may be combined to form **software system architectures**. Within the scope of this thesis these software architectures then become **software configurations** once instantiated with data related to a specific software system. Each component group has a number of **attributes** associated with it, and may be recorded on identification sheets or **proformas** as they are developed or progressively re-identified during the **ISCM process**. These component groups are then amalgamated into configurations at varying levels of abstraction and documented as such. Additionally, as knowledge reclaimed about a system configuration is incremental in nature and systems by definition do not remain static, the ability to update and store configurations at determined points in time is required. This is one of the roles of the **Extensible System Information Base**.

The above features of the ISCM process will be discussed in detail in Chapter 4. The first stage however, to enable modelling of software system architectures within the ISCM process, is to identify the component parts or building blocks of software architectures and the attributes of each of these component types making up the architecture.

3.3.1 System Building Blocks

As a starting point for any software configuration management system it is important to establish a form of representation for the elements of system modelling. Terminology and defined functionality of tools are not currently consistent, and are generally determined by the conceptual approach taken by individual researchers and developers [136, 224]. For example, although software 'object' is a commonly used term, other tools or systems may refer to 'elements' [336, 394], 'items' [280], or 'modules' [378]. The following section describes a number of current views on what constitutes and contributes to a system model. More extensive discussions may be found in [81, 144, 376, 377].

3.3.1.1 Software Objects

A software object or configuration element [141] can encompass any kind of identifiable, machine readable document produced during the course of a project and as such they form the fundamental building blocks of any system [376]. Examples of software objects are program code, documents, command files and test data etc. Most systems define a software object as having a 'body' containing the information, and some form of 'unique identifier', together with an associated set of attributes to define author, creation time, last read access etc.

Software objects can be further refined and described in relation to how they were created and also with regard to their internal structure [376]. The main distinction between software objects is whether the object is rederivable or not [377]. Source objects are those objects that are created manually and by definition they are provided by human input. These objects must be preserved as they have a uniqueness related to the definition 'manual' and cannot therefore by automatically regenerated if altered or deleted. Heimberger [190] refers to source objects synonymously as 'non-rederivable' objects or 'components' and Conradi [101] uses the term 'primary objects'.

Derived objects are those objects which are created automatically within the system by programs or tools called 'derivers'. Such objects need not necessarily be stored within the system as they can be rederived, providing that the corresponding source and derivers are still in existence. In practice though, most configuration management systems will maintain a cache of current derived objects to avoid object regeneration delays, which can be considerable [376].

There is some controversy over objects that are derived but which then require manual inputs (e.g. program templates), about whether they should be classified as source or derived objects or indeed even if they should be split into two components [376]. This point is exemplified by the views expressed during the plenary discussion on software objects at the Software Version and Configuration Control Conference [377].

The second distinguishing characteristic of software objects is whether they have bodies that are 'atomic', 'structured' or occasionally 'bodiless'. The latter category arises with the need to represent a particular version of software that requires control but which cannot be held or represented as a body of an object e.g. versions of host operating system, or to represent objects that cannot yet be created [8]. Atomic objects cannot be further divided into smaller objects, whereas structured or 'complex' [29] objects consist of an hierarchy of sub-objects with atomic objects at the lowest level and connected by relationships. A configuration is a type of structured object.

3.3.1.2 Configurations

In its simplest form a configuration may be defined as 'a list of modules of which the program is composed' [23]. Krane [377] and Venkatrami [385] extend this definition to include all information required to recreate a system (sources, tools, parameters etc.). Venkatrami further defines a configuration as a collection of versioned objects, where the collections themselves may be versioned, and others describe a configuration as a set of constituent software components, together with their control and communication interconnections [359].

More specifically, Estublier defines a configuration as 'a consistent set of objects, one for each module related by their dependency relation' [136]. This definition highlights the difficulties associated with loose coupling of terminology and functionality and of using generic definitions. Many tools have a tendency to describe 'module' in a different way: for example *make* [146] has no concept of a module it relies only on files; *Cedar* [369] splits each module into an 'interface module' and an 'implementation part; *Gandalf* [295] has a similar module concept but adds a set of revision to each version; and *Adele* [135] goes one step further by representing each interface by a set of 'views'.

Tichy [376] describes two sub-components of configurations: sequences and composites. A sequence is a list of object and/or version identifiers of objects, each of which have the same type, perform the same function and can be treated in the same way for SCM purposes. A typical example is a list of library object modules. In comparison a composite is an object analogous to a record structure comprising a fixed number of fields, each consisting of a field identifier and a field value (object identifier or version group identifier). In this case each field does not perform the same function, e.g. a software package consisting of a program, a manual and a set of test programs.

Clemm [92] in his description of the *Odin* specification language also distinguishes between atomic objects edited directly by the user, and derived objects automatically produced by a computer program or tool. He further defines derived objects as an increasingly complex series cumulating in composite derived types which are analogous to Tichy's composite objects.

3.3.1.3 System Model

There is currently considerable debate about what aspects of a system should be described in a system model. Generally a model will not completely identify a system configuration, rather it provides a framework to enable the identification, comparison and selection of a number of configurations [81]. Tichy defines a 'generic' or 'loose' configuration as a system model. A system model defines what elements are needed in the configuration but then requires version selection procedures to select particular revisions / variants in order to produce a definitive baseline. It is this definitive baseline which has the ability to completely and unequivocally describe the system. Similarly, Conradi [101] uses the terms 'unbound' and 'bound' respectively to describe generic and baseline configurations. In general, the structural description of a configuration can be represented by graphs, whilst the lowest level components are represented by atomic text objects. Schwanke extends the definition of a system model to include construction information [377].

Generic configurations make possible the compact representation of a large set of possible baselines, without having to enumerate all the possible combinations. An alternative approach is to maintain configuration tables but these are bulky and difficult to maintain for large systems.

Sacchi [342] proposes a six-dimensional information model which complements Tichy's system model. It provides reasoning on which the construction of system configurations may be based, these six dimensions are: organisation, structural, spatial, temporal, purpose and procedural. Currently no system supports all these information dimensions but the adoption of knowledge based techniques, object oriented techniques or entity-relationship models may eventually aid in this process. A more complete discussion on system modelling may be found in the plenary discussion on the International Workshop on Configurations and Version Control [81].

3.3.1.4 Discussion of System Building Blocks

From the previous section it can be seen that there is a great deal of instability over the terminology relating to the components of a software system and how they should be described and combined into configurations. Additionally, the changing nature of software systems towards embedded, 3rd party and bespoke amalgamated applications, and those with a proprietary GUI front end raises a number of interesting issues concerning defining the boundaries of applications and where maintenance responsibility lies for systems that are a mix of hardware, bespoke and 3rd party software. Additionally, the issue arises as to whether distinctions should still be made between derived and source objects, since many of the derived objects in a system may not be re-creatable if they are supplied as part of another application. Alternatively, they could be thought of as primary or secondary derived objects depending on whether they originate as part of the host system or whether they originate from another vendor. The work in this thesis will therefore make some assumptions and definitions in Chapter 4 about the status of objects or software configuration items within systems, the attributes of these components and the roles that they play in combining to form software system architectures as defined by their associated system models.

3.3.2 Methods for Modelling Software System Architectures

Within both development and maintenance the issue of creating architectural descriptions or system models of software systems has become an increasingly important area of research and development [161]. This has been brought about by the realisation that, to date, much of the architectural modelling of systems has been done on an informal, ad hoc basis leading to unusable, eroded and unmaintainable architectures which are characteristic of legacy software systems [282, 313]. Mechanisms for modelling software system architectures have however, been developing progressively over the past two decades such that areas of research now encompass those of graphical design notation, module interconnection languages (MILs), templates and frameworks for specific domains, architectural patterns and formal models for component integration.

There have been numerous approaches to the modelling of software system architectures, predominantly based on the use of interconnection models and description languages. An interconnection model is an abstract description of the components in a given domain whilst a description language is the structured syntax and semantics required in order to express the attributes of the components within that domain [396]. One of the most descriptive approaches which has stood the test of time is that of the Module Interconnection Language (MIL). MILs were first introduced in 1976 by DeRemer and Kron [126] as a means of describing the interconnection of software components, in what was subsequently termed the field of programming-in-the-large (PITL) as opposed to programming-in-the-small (PITS) which is the activity of producing the individual components. Since their inception other MILs have been developed such as the configuration languages of *Mesa* [198], *Conic* [353], *reuse-oriented MIL* [317] and more recently an *object-oriented MIL* [180] for connecting the components of object-oriented systems together. This section introduces the key concepts of interconnection models in general and surveys a number of MILs and related approaches.

3.3.2.1 Interconnection Models

Interconnection models have been used to support the management of system evolution. An interconnection model essentially consists of two sets, a set of objects (the components of the model), and a set of relations that define the interconnections that exist amongst the objects in the model. The model is represented by a graph structure with nodes as objects and arcs as relations [310].

The unit interconnection model is the most basic, it defines dependency relations between the files or modules comprising a software system. This model supports modular construction of software and may be used to aid compilation and recompilation strategies, change notification and system modelling.

The syntactic interconnection model is a finer grained model describing relations among syntactic elements of a programming language i.e. the objects within the modules such as procedures, functions, types, variables etc. (cf. files and modules as in the unit interconnection model). Whilst the unit interconnection model can only indicate the general location of changes, the syntactic model indicates the syntactic objects that have changed. This model may be used for change management, smart recompilation, static analysis and system modelling. In systems using this model e.g. *Gandalf* [178], both revisions and variants may be identified, and use may be made of a programming-in-the-large language, in which versions are understood as part of the system description language. Syntactic consistency checking can be performed automatically by the system.

The semantic interconnection model goes a stage further by trying to capture and express how the objects comprising a system are meant to be used and why. The *Inscape* [310, 311] environment is using an input/output predicated approach to define the semantic interconnection model and interconnections between objects. *Inscape* provides a module interface specification language, *Instress*, to describe the properties of and constraints on data and the behaviour of operations. In this way *Inscape's* version control mechanism, *Invariant*, provides a formalisation of version control that has some significant advantages over version control mechanisms that use either the unit or syntactic interconnection models. These advantages can be summarised as computability and checking of system models with respect to their semantics as well as the syntax. Perry [310] deals in some depth with the concepts and uses of interconnection models.

More recently, the *3C* and *REBOOT* models have been developed [396] primarily as mechanisms for identifying components for reuse in the systems engineering domain but also with the potential for describing software configurations. The *3C* model is a prescriptive model of the attributes that a component should embody. It encompasses three key aspects namely:

- Concept (an *abstract description of what the component does*)
- Content (a *description of how it achieves its purpose*)
- Context (a *description of the domain applicability of the component and its interactions with other components.*)

In contrast the *REBOOT* [396] component model is a classification model based on descriptions of the component and encompassing:

- Dependencies (*relationships with other components*) .
- Abstraction (*object that the component implements*)
- Operations (*operations that the component offers*)
- Operates on (*interaction with the environment*).

Similarly, the features it offers have potential for inclusion in models for describing software configurations.

3.3.2.2 Module Interconnection Languages (MILS)

A module may be described by an interface that specifies the resources provided and required by the module, and by a body that details how the module provides the resources that it should. This definition enables the distinction between programming-in-the-small (PITS) and programming-in-the-large (PITL) to be made. PITS is concerned with the development of an individual module in a software system, whilst PITL is concerned with the interactions between modules of a system [126, 132]. Perry [311] defined four interrelated aspects of PITL:

- The description of module interfaces.
- The control and implementation of variants of these interfaces.
- The modelling or configuration of a system from its components.
- The generation of a system from its model or specification.

Configuration control is therefore essentially a PITL concern. MILs work at the PITL level and several MILs have been proposed to describe the evolution of software systems. Some of these MILS and the systems that use them are described below but a more comprehensive treatment of the subject's foundation is given in Prieto-Diaz and Neighbors paper [318].

The concept of MILs was originally proposed by deRemer and Kron [126] to support PITL. This expressed how configurations could be constructed from their constituent parts or 'modules', by formally describing the interdependencies existing between components of a system [318]. The first such description language was *MIL-75*. Subsequent research has concentrated on extending this principle to include mechanisms for describing system evolution. Two notable examples of MILs are Cooprider's *MIL* [105] and Tichy's *INTERCOL* [371, 372]. Cooprider's system integrated a MIL with a version control system. Tichy's work improved on Cooprider's through the provision of a facility for automatic verification of interface consistency among separately developed components, and the ability of his system to determine which version of which component should be used to form a particular configuration. The results of Tichy and Cooprider's research are to express software systems as families of related systems, and subsequently to introduce the concept of module and sub-system families, each member of a subsystem or module family being an 'implementation' or 'version' of that family.

Several programming environments have evolved that provide module interconnection facilities. *Gandalf* [178] is a software development environment whose *System Version Description and Generation Facility (SVDE)* incorporates Cooprider's version control system and Tichy's *Software Development Control Facility (SDCF)* [372]. The *Mesa* [231] system supports modular program development through a configuration description language, *C/Mesa*, which specifies how separately compiled modules are to be bound together. Similarly, the *Cedar* [369] environment uses descriptive files to list the files of a system and a system modelling language derived from *Mesa* to describe the interactions between these files. However, widespread use of these two systems is hindered by their language and machine dependence.

Module interfaces in the above systems are determined by the syntax of the resources they require and provide, that is, all implementations share exactly the same interface. Narayanaswamy [286] argues that functional properties are a more accurate way of defining members. The *NuMIL* environment therefore takes the view that each member of a module family should satisfy the same abstract interface specification and defines this specification as a module/subsystem family template for each family. They need not however, necessarily share the same syntax. Another important contribution of this work is the treatment of upward compatibility of modules and systems based on both syntactic and functional properties.

Perry [310, 311] in the *Inscape* environment has also explored the use of formal module interface specifications to describe the semantics of data and operations used by a system environment. The MIL used is *Instress* which introduces the notion of obligations to describe operational results. These complement the descriptions of data properties and operational behaviour to describe operation results obtained by using Hoare's [195] pre- and post-conditions. *Inscape* also provides a formalisation of the notion of version equivalence and presents four different kinds of compatibility: strict, upward, implementation and system.

Another MIL based system, *CRUISE* [362], proposed a rigorous representation scheme for the structural evolution of software based on the notion of a hierarchy of interfaces. Through the use of a MIL which describes the semantic properties of the interface, it enables descriptions of architectural designs for software systems as well as attribute information that facilitate identification and retrieval of configurations. Feedback mechanisms give the impact of interface modifications on the integrity of software systems.

A number of configuration description languages have also been defined which enable formal descriptions of components and their architectures to be made and translated into implementation languages via application of formal transformations. Examples of such algebraic specification languages include the *Library Interconnection Language (LIL)*, *ACT TWO* and *Π*. *Meld* and Configuration Description Language *(CDL)* are language-independent object-oriented design level languages. They too are formally defined and fulfil similar roles to *LIL*, *ACT TWO* and *Π*. A detailed discussion of these languages, their properties and usage can be found in Whittle [396].

Yau [403] has investigated the use of AI techniques, based on first order logic to interpret the interconnection behaviour among the components of a software system. This allows automated reasoning techniques to be used for validity and integrity checking of software interconnections after modifications have been made. Similar logic techniques are being used to assist in the automatic generation of expert or knowledge-based systems that incorporate 'machine-learned' human domain knowledge about computer systems [116].

3.3.2.3 CONFIG language

Winkler [398] proposes an approach based on representation of version information as part of the programming text. This approach describes different configurations of modular programs using constructs that are integrated into the programming language. He proposes that there should be a CONFIG part of a program building block (module, compilation unit) in which the programmer can express, firstly, to which versions the building block belongs, and secondly, which versions of other building blocks it uses. This adding of constructs to express versions as pairs of revisions and variants, will enable the description of configurations of program families in languages such as Ada and Modula-2 which already have constructs for PITL. Winkler uses the term 'pragmatic configuration' to define this description of version information within the programming language [398]. Pragmatic configurations effectively extend the principles of syntactic configurations and semantic configurations in which the programming language describes the syntax and semantics respectively of the interface building blocks. It is this respect that distinguishes Winkler's approach from most others that are oriented towards languages such as C or Pascal, or that contain their own framework for modularisation e.g. *INTERCOL* [372] or *NuMIL* [286].

3.3.2.4 Proteus Configuration Language

Proteus Configuration Language (PCL) is a MIL-evolved language designed to model the architecture of multiple versions of computer-based systems [360]. *PCL*, has two primary functions. Firstly, it can be used to describe, at an abstract level, the architecture of different system versions. Secondly, through specification of component dependencies it acts as a configuration language and the basis of a systems building and version management system. *PCL* models a system around the basic entity types of:

- Family entities (*hardware, software and documentation*)
- Version descriptor entities (*specific attributes of a single version of the system*)
- Tool entities (*tools used to build the modelled system*)

The basic facilities are then extended and the model linked to specific design methods using:

- Classification definitions (*used to classify entities in a number of different dimensions*).
- Relation definitions (*used to define relations between the family entities and other family entities, version descriptor entities and tool entities*).
- Attribute type definitions (*used to define attribute types as an enumerated set of identifiers*).

One of the key features of the *PCL* is the support offered for variability in system families such that it supports structural, implementation and installation variability. To achieve this *PCL* models system families as a set of stable and variable parts. The stable parts of the system family can be defined in base or 'ancestor' components and then inherited by specific components, whilst the exact nature of the variable parts are made explicit by version descriptor entities and then specified using conditional inclusion.

3.3.2.5 DOCMAN

DOCMAN [58] is an intermediate file format language developed by British Telecom Research Laboratories to enable automatic processing of system information. The focus of the DOCMAN file format is on portability between hosts and ease of modification and manipulation of the system information contained within the file. Therefore, in marked contrast to the algebraically specified configuration description languages outlined in section 3.3.2.2. the DOCMAN file consists lexically of a sequence of text lines and semantically as a set of ordered records and nested sub-records indicated by simple coding mechanisms. The file appears as a left justified sequence of lines which although not aesthetically pleasing can none-the-less be deciphered by humans and readily parsed by programs whose role it is to extract particular data from the file.

3.3.2.6 Domain Specific

Another approach primarily aimed at the specification and construction of software systems from reusable components has been proposed by Neighbors [290]. This approach makes use of a knowledge-base to construct systems specified according to the *Draco* methodology [289]. The *Draco* methodology is concerned with the modelling of systems through the combination of information-capture by an 'application domain analyst' who understands how systems of the required type are constructed, with the appropriate computer science modelling techniques known to a 'modelling domain analyst'. Fusion of these respective top-down and bottom-up approaches to information acquisition is then achieved in a more detailed format via a 'domain designer' who specifies the resultant problem domain to the associated tool. Individual systems within the problem domains known to the tool can then be specified at later time-intervals by a systems analyst who is not necessarily an expert in the particular domain area. This approach ultimately results in a number of domain languages being produced. Each language describes the components of a system within a particular problem domain in a syntactical language and constructs pertinent to that domain. Tests and results to date have shown this to be an effective mechanism for:

- Improving understanding of the parts of the system within a particular domain.
- Checking specifications and requirements for a system in that problem domain.
- Educating people in the organisation with the general structure and operation of a system with that domain.
- Deriving working systems from the requirements stated in domain specific terms.

Domain-specific Software Architectures (*DSSAs*) are also being developed for use in adaptive intelligent systems (*AISs*) with regard to [188]:

- Meeting functional requirements.
- Decomposing expertise into highly reuseable components.
- Selecting relevant components for automatic configuration into an instance of the specified architecture.

A *DSAA* comprises a framework for a significant number of applications, reuseable 'chunks' of domain expertise and a method for selecting and configuring components within the architecture to meet particular application requirements. *AIS* applications have particular problems associated with the configuring of systems due to their requirement to perceive, reason and act to achieve multiple goals in dynamic, uncertain and complex environments. This approach whilst also placing emphasis on the importance of domain analysis differs from the *Draco* approach in trying to generalise expertise and recognise common architectural properties across diverse domains rather than create a specific domain language for each domain area.

3.3.2.7 Rapide

Rapide is an event-based concurrent object-oriented language specifically designed for prototyping architectures of distributed systems [247]. *Rapide* uses the architecture of a system to provide a global view of how a set of object-oriented components have been combined into systems. Architectures within *Rapide* consist of a set of:

- Interfaces (*specifications of modules*)
- Connection rules (*define communication between interfaces*)
- Formal constraints (*define legal/illegal patterns of communication*)

Rapide differs from other event-based languages, such as *LOTOS* for modelling communication protocols and *Esterel* for modelling synchronous systems, in its ability to explicitly represent dependencies between events [247]. In modelling dependencies *Rapide* shares some common ancestry with hardware description languages but it adds the capability to model dynamic as well as static systems. Another distinctive element of the *Rapide* approach to modelling systems is the capability of the architectures to be validated, executed and performance-monitored prior to the system being built. This is achieved through a range of automatable analysis techniques ranging from execution and simulation to runtime constraint checking and formal proof. These architectures can then act in a number of ways, for example, to monitor system development, to stimulate and analyse the behaviour of dynamic systems and to act as reference architectures in order to assess compliance of systems to industrial standards.

3.3.4 Discussion of Architecture Description Languages

Section 3.3 has briefly outlined the developmental history of languages for modelling software system architectures and has described the key properties of a representative sample of such languages. This section selects and justifies a number of these features for inclusion in the definition of the Inverse Configuration Description Language (ICDL).

A number of languages have been defined as a means of syntactically and semantically expressing the components and relations defined within associated interconnection models. Much of the seminal work in this area has been in relation to MIL development and in particular has stemmed from that of DeRemer and Kron in the mid 1970's [126]. The advent of MILs enabled a distinction to be made between programming-in-the-small (PITS) and programming-in-the-large (PITL) whereby PITS is concerned with the production of an individual component (module) of software whilst PITL supports the combination of these components into complete systems.

However, whilst MILs are very effective at defining the module interfaces and the resultant construction of software systems they are not concerned with a number of factors which play an important role in enabling a maintainer to conduct efficient and effective program comprehension. MILs for example, are not concerned with:

- What the system does (*specification information*)
- The business organisational aspects of the system (*analysis information*)
- How individual modules implement their functions (*detailed design information*)

Indeed, each of the above features may be considered essential to the program comprehension process as indicated by the reiteration of the three key areas encompassed within the program comprehension activity:

- Understanding what the system does and how it related to its environment (*specification and analysis information*).
- Identifying where the system change should be effected (*analysis information*).
- Deducing how the components identified for correction or modification work (*detailed design information*).

Another deficiency apparent in many of the early MILs is their sole concentration on software description resulting in inadequate provision of facilities for describing the associated hardware or document structure. Some early MILs also restricted their modelling capabilities to those of single versions of systems. Languages and systems have however emerged to address the issues of handling multiple versions of software or program families, for example the initial work in this area by Tichy [372] and Cooprider [105] and most recently that of Sommerville [360]. The PCL language of Sommerville and the REBOOT model referred to by Whittle [396] also begin to address the issues associated with widening the scope of system modelling to incorporate the environment and components such as the hardware, enabling tools and associated documentation. It is in these respects that the PCL work most closely resembles that of the ISCM modelling process.

Architecture description languages to date have also tended to concentrate on describing the static structure of systems and not their dynamic characteristics. Dynamic modelling may be considered from two perspectives: firstly that of being able to dynamically reconfigure a software architecture according to specified build patterns; and secondly as a means of modelling the dynamic behaviour of systems. The latter area, whilst likely to be an increasingly important area of research due to expansion in the number of event-driven and real-time systems is considered to be outside the scope of the thesis. Rapide, however, is developing as a language capable of simulating and modelling the run-time behaviour of such systems. The dynamic reconfiguration of software systems is another area in which PCL is making some grounds [360] through incorporation of inheritance and conditional inclusion.

A number of configuration languages are also being developed which formally define the components of a software system with the primary aim of enabling and verifying the correctness of system builds. Whilst the merit of these languages is noted, their aim is also considered to be outside the scope of this thesis which is concentrating on identifying and documenting the composition of existing systems rather than building new systems from component parts. For these reasons they will not be considered further. In marked contrast to these mathematically specified languages, the DOCMAN approach to describing software system configurations is based on a simple text file and simple combinations of characters to represent commands. At this stage in the work, the portability, flexibility, learnability and ease of use characteristics resulting from the DOCMAN format are considered more applicable to the ethos of the ISCM process than the ability to formally verify the built configuration.

Another important concept that has relevance to the ISCM method is that of capturing and making use of domain expertise and knowledge. Models such as *3C* [396], methods such as *Draco* and frameworks such as *DSAA* all rely on being able map knowledge and problems to their relative domains in order to model system architectures, either specifically as in *Draco* or more generically across application domains as in *DSAA*. However, from a program comprehension perspective the information contained within the *3C* model describing what each component does; how they achieve this; their interactions with other components and their domain applicability, perhaps offer the most potential for exploitation.

Taking into account the combinatorial effect of the features described, modelling within the ISCM process may be considered to be a natural extension to earlier work in the areas of PITS (construction and modelling of single components) and PITL (construction and modelling of software systems). To this effect the ISCM concept of programming-in-the-environment (PITE) is introduced. **PITE may be defined as the construction and modelling of entire software environments**; where environments may be taken to mean the modelling of components belonging to software other than that of the application, hardware, documentation, tools, cognitive, configuration and version information etc. The underlying models of PITE are based on the configuration (architecture) abstractions that will be defined in Chapter 4. Correspondingly, the syntax and semantics required to express the attributes of the components modelled within these architectures are provided by the Inverse Configuration Description Language (ICDL). The ICDL language constructs are also described in Chapter 4.

3.4 Software Manufacture & Configuration Creation Process

During development or after maintenance configurations must be rebuilt from their component parts, this is termed software manufacture or system synthesis. Essentially the configuration creation or manufacture process may be spilt into three stages:

- The choice of a set of groups which are to contribute versions to the configuration.
- The selection or binding of an actual version from each of the contributing groups.
- The building of the configuration according to the relationships between components and translation rules defined by the dependency graph.

The primary emphasis of ISCM is on the cognitive aspects of understanding a system configuration rather than on the physical activities associated with rebuilding it. However, as both processes involve study of the associations between component parts, it should be possible to draw a number of parallels between the system composition process of traditional SCM and the system decomposition activities of the ISCM process. Krane [224], for example, views the entire configuration creation process as the transformation of an incomplete high level specification of a software product into a complete high level specification capable of being executed. In his model he represents this process by a developing series of attributed graphs, such that nodes represent objects which may or may not have associated attributes, and edges represent relationships existing between the components. Translation processes can be defined to add nodes and edges to the graph or to propagate values to the nodes. In this respect the ISCM process holds a similar view in that it too moves from an incomplete to complete high-level specification of the system, albeit from the inverse angle. The remainder of this section therefore briefly surveys the synthesis process and formulates some conclusions about the applicability of such activities to the ISCM process.

3.4.1 Component Group Selection

The first stage in the configuration manufacture process is to produce a list of the component groups required to donate versions in the construction of the product. There are several mechanisms for conveying this dependency information to the configuration management tool. In most language-dependent environments such as *Cedar*, *Gandalf*, *Ada* and *Modula-2*, the dependency relation is explicit in the syntax of the language, and is provided to the tool in compilation. Information may also be extracted directly from the source code by parsing the source code for constructs that specify the imported files or modules. Adele uses this mechanism as does *mkmf* - the program that produces *Makefiles*. *CCC* uses two scanners, one that detects user-specified strings in source code, and a second that detects definitions/references in object code [357]. Alternatively, module interconnection languages may be used to confer the dependency information to the tool.

The result of such mechanisms is typically the production of a version independent or generic configuration of a system. Tichy [376] for example, describes the generic configuration by means of an AND/OR graph model, using leaf nodes to represent atomic objects, AND nodes to represent configurations, and OR nodes to represent source or derived version groups. Conradi [101] uses an AND-view/OR-view graph representation for unbound (generic) configurations.

The *Make* system model explicitly defines the exact configuration of a system. However in most SCM systems the generic configuration will yield many specific configurations, only some of which will be viable. Selecting the right modules and variants without exhaustive search is one of the most significant problems in configuration management [136, 376]. Mechanisms must therefore exist to select a specific version from each of the defined component (version) groups whilst still maintaining the integrity of the system.

3.4.2 Version Selection

The typical algorithm for component selection is to start at a particular graph node (component group) and select from each component group in turn, one of its variants/revisions. This may be considered a 'binding' process as attribute information such as variant/revision identifiers and physical locations are being bound to the generic graph. The set of selected variants/revisions constitutes the baseline or bound configuration [136].

Each tool has its own methods of selecting specific versions from the generic configuration. Some of these are more flexible than others, the flexibility depending to some extent upon the nature of binding between objects in the dependency relation, for example the dependency relation may specify:

- Exactly which revision and version to use, the tool provides no selection
- The variant to use, the tool selects the revision
- Only an interface, the tool selects both the variant and the revision.

Selection is generally subject to a given set of requirements or constraints which are applied to the generic configuration. Many tools base selection on syntactical constraints, for example, revision number, state and author are often used to select revisions. A variant may be based on some environmental factor such as the hardware platform being used.

The dependency relation of *Make* is very strict, defining exactly which files to use and allowing no revision or variant selection by the tool. To use any files other than the most recent version requires that the dependency relation in the *Makefile* is explicitly changed. *Shape* [253, 254] - an enhanced *Make* program, offers some improvement by including support for configuration rules. These rules control the selection process for component objects during identification, build and rebuild of system configurations. Configuration selection rules in *Shape* consist of a sequence of alternatives and associated predicates, and succeed if one of the alternatives succeeds. The configuration rules also support variant selection through the use of variant flags that may be passed to the transforming tool [254].

DSEE [236] uses configuration threads to define which revision to select. A configuration thread is a rule based description of the version of components to use for a particular build of a system. In *Odin* [92, 94] a query corresponds to the configuration thread in *DSEE*. A query is a request for a particular derivation. Queries can be obtained from files or presented interactively, and a history mechanism allows modification and reuse of interactive queries. In *Jasmine* [258] an image is equivalent to DSEE's bound configuration thread.

Gandalf [178] assigns defaults for selection of revisions which may be changed dynamically by other modules, while *Adele* [135] selects revisions using constraints on attributes. *Adele* also allows selection of variants using constraints which rely on semantic properties of the variants and not just on names and dates. The selection process continues through the alternatives until constraints are matched. Adele also differs from systems such as *Make*, *DSEE* and *Odin* in which the relationships between components of a given system are statically defined in the system model. *Adele* takes a dynamic approach by using the desired characteristics of a system to derive the configuration rather than a composition list. The description is evaluated against a module dependency graphs and the module descriptions stored in a database of programs. Dynamic selection means that it is possible to obtain several different compositions list for a system.

Clemma [333] supports both static and dynamic construction of configurations. Static construction is via a list of components and their versions, while dynamic selection is achieved through querying the relational database. *Gypsey* [95] also used queries, first to select a subset of versions that satisfy some predicate and then a specific version that satisfies some selector. In the *Darwin* [277] environment, version selection is based not only on the objective requirements, that is, the syntactic and semantic constraints that emanate from the structural aspects of the system, but also on subjective restraints which are imposed explicitly by the users, programmers and managers involved in the software development process.

Module interconnection languages use a different approach to version selection, concentrating instead on the interfaces among software modules. Most *MILs* can represent versions of the implementation part of the program but have difficulty in representing versions of interfaces. Exceptions are *Mesa* [164] and *Cedar* [369] which use the *C-Mesa* sub-language for describing configurations.

3.4.3 System Building

The final stage of the system construction process is to generate the derived objects and executable image of the system from the records of versions, sources and tools held or defined within the system model. Building occurs according to the dependencies, translation rules and commands in the dependency graph. These rules may describe how to generate a particular target file, i.e. which kind of source object is transformed into which kind of derived object, how the transformation is performed, and the parameters used. Essentially, when a transformation rule is evaluated the name of the source objet is passed to the current selection rule and bound to a concrete source object in the object base. Alternatively, commands may specify the extraction of a particular revision from the archive for the purposes of linking and compiling [275].

The build tool or facility should have the flexibility to support construction of products consisting entirely of current versions of components, and of products constructed from versions reconstituted from archives. *SERS* [329] for example, uses stocklists to specify the files, versions etc. of components which form a release, and generates a corresponding loadlist which lists those files that require recompilation. *Shape* [253, 254] uses a configuration identification document (CID) which is essentially a *Shapefile* that is completely bound specifying version number, variant identifier, tool versions etc.

To maximise efficiency it is important to make the building process as fast as possible. It is generally impractical to rebuild an entire system every time a component in the system is changed. Various mechanisms therefore exist to avoid complete rebuilds such as caches of derived objects [236], timestamps [146], opportunistic processing [212, 213], smart and smarter recompilation [374, 375]. This subject is outside of the scope of the thesis and as such will not be treated any further here, McCrindle [263] offers a more comprehensive review of the different recompilation strategies.

3.4.4 Discussion of the Configuration Reclamation Process

In order to completely recover the system architecture/configuration of a legacy software system the ISCM process must incorporate similar activities as those described for the construction of software systems. For example, there is the need to identify the version groups from which components are to be selected; to identify the correct revision or variant of the component from within the version group; and to determine the parameters or attributes attached to particular component that govern the combinatorial process. There is also the need to understand or recover as much syntactical, semantic and cognitive information associated with a component as possible in order to build the configuration correctly (forward process) or to document the configuration accurately (ISCM process). Conversely, there are also a number of fundamental differences between the forward and inverse processes in that for legacy systems the process tends to be much more iterative rather than sequential in nature. The end-product is also different in that the SCM construction of a system configuration involves moving from a system description to an executable system, while the ISCM identification of a configuration moves from an executing system to a description of the component parts of the system and relationships existing between them. Factors of both a similar and dissimilar nature were taken into account when modelling software systems within the ISCM process. Additionally, it was found that as maintenance proceeded over the lifetime of a product the configuration description document could be built-up and controlled as part of the legacy system and hence the two approaches began to coalesce. These issues are discussed further in Chapter 4.

3.5 Storing Software Configuration Architectures

The underlying mechanism of the ISCM process and its associated method is the representation of the fully- or partially-reclaimed software systems architectures by means of an Extensible Systems Information Knowledge Base (ESIKB). Within the scope of this work the characteristics of a number of Unix, MS-DOS and Windows95 systems and their applications were studied (see Chapter 6) and architectural models manually built-up for each application. The characteristics of the separate models were then used in combination to create the generic model; to determine the type and structure of information to be stored; and to identify a set of initial rules for reclamation of this information. These issues leading to the creation of the ESIKB will be discussed more fully in Chapter 4, whilst this chapter surveys a number of possible storage mechanisms and organisational formats for the resultant data.

3.5.1 Configuration Information Storage Mechanisms

SCM systems must all have some underpinning file or database structure within which to store the various software products and configurations [190], this may be termed the 'object' or 'information' base. The hierarchical file structure of the underlying operating system is the traditional way of representing objects. Objects are stored as files, and their properties are recorded in another file, e.g. *Make's Makefile* and *Odin's ODIN* directory. They have an acceptable level of reliability, however they have limited recovery procedures, consistency control, access synchronisation and authorisation [376]. The *Team One* [385] configuration management system is based on an extension to a normal operating system directory. It resembles a normal directory but in addition can freeze a directory structure at discrete points in time which it calls checkpoints. The *Shape* toolkit uses an *Attributed File System*, *AFS*, realised as a callable interface, to enhance the regular *Unix file system* by increasing the number of attributes that can be associated with a file. [253, 254]. *Gypsy* is built on an object oriented extension of *Unix* that provides mechanisms for customising directories with extra commands for inserting and deleting components, altering file projects and listing directories etc.

Commercially available databases may be used as the underlying structure for the object base, for example the *Changeman* system [395] has the *Oracle* relational database at its centre and *Galileo* uses *Ingres* as its basis. Database Management Systems (DBMSs) provide the high reliability and systematic mechanisms necessary for handling recovery, consistency control, access synchronisation and authorisation. They also offer facilities that promote data sharing, can efficiently store a large amount of information and support retrieval and concurrent access by multiple users [40]. However the nature of SCM requirements do not parallel those of commercial DBMSs. SCM requires processing of moderate numbers of large objects with complex internal structures in contrast to the business oriented applications which require processing of large quantities of much smaller records [376]. The result is that for SCM purposes these databases are often inefficient, inflexible, inactive, unable to represent deltas, and do not provide enough modelling support. In addition most have difficulties in representing versions and revisions of a single object. There may also be difficulties in interfacing to different languages, tools and hardware [40, 134, 136].

Both the configuration librarian, *Clemma* [333], and the integrated system, *PCMS* [280], are based on relational database models. *CCC* [356] and *Lifespan* [378] are two commercial products which have developed custom databases optimised for configuration management activities. *Eclipse* is built on the *SDS2* [9] database system, that has been designed specifically to support software development by large teams. It employs a binary semantic data model with bi-directional links and supports a wide variety of data types in addition to allowing other types to be defined by the user. *SDS2* is based on an extended entity relationship model and is designed to serve as a universal data manager for a large set of tools within an ipse. The possibility of using this database instead of the Unix file system is being considered by the developers of Shape. *DAMOKLES* [127] is another example of a dedicated software engineering database.

Attention is now being directed towards object-oriented models and entity relationship models for object base support. This is in an attempt to yield more accurate data models and increase efficiency. Bernstein [40] recognises this as being at two levels, that of layering new facilities on top of existing relational database systems, and that of creating new DBMS architectures specifically targeted for design operations. The layering approach is unlikely to be as powerful or efficient as the new database architectures, however the new architectures are still in their infancy.

Perspective Kernel [220], a product from System Designers and an integrated database from *Datev* have entity relationship approaches at their centres. Similarly, *Galileo* [336] uses an entity-relationship-attribute data model represented in an underlying relational datbase. By optimising the database for retrieval rather than for updating acceptable response times have been achieved [336]. The *Common Lisp Framework* [92] and the *EPOS* software engineering environment [101] are partially based on object-oriented approaches. *Marvel* [212] has an object base that is conceptually related to object-oriented programming languages in that each object is an instance of a class that defines its type. However, unlike most object-oriented languages, *Marvel's* object base is persistent, that is, it retains its state across invocations of the environment thereby providing a file-less environment. EBDLOG [20] is a prototype logic database system implemented in *Prolog*. It allows a database to be defined as a logic programming unit and definitive Horn clauses. A more detailed discussion of approaches to the use of DBMS to support software engineering is given in [40].

The above discussion indicates that there are a number of possible storage mechanisms for the physical representation of the configuration items (components) and their corresponding configuration descriptions. These can be broadly summarised as:

- ***Flat-file***: this is the simplest form of storage with information being represented as a sequence of bytes. The advantage of this method is the universal usage of this form of file storage in all operating systems and hence there is no extra overhead. However, issues may arise concerning the modelling of shared data items. Within this file organisation data are normally stored in records of fixed or variable length. Each record generally represents an instance of an entity type and the files contain the entity values. Within a file each record may be organised in a hierarchical or network structure, each node corresponding to a class of real-world entities and an edge representing the links between the entities [121]. Permissible operations are the insertion of new records, deletion and modification of record types. Primary and secondary keys are needed for sorting and searching and an end of record indicator is required for separation of individual records.

- ***Relational***: relational databases are currently the dominant database technology [82] and represent an elegant and powerful way of viewing and manipulating data at a logical level. The relational storage model allows data to be represented in the form of tables whose size is predefined. The model originates from the mathematical concepts of relational algebra and set theory. Within this approach all information is represented logically by values in tables. Every data value is logically accessible by a combination of table name, column name and primary key value. A relation may be viewed as a two-dimensional table representing an entity set. The relation has a fixed number of named columns, or attributes, and a variable number of rows or tuples. The overall relational schema contains a collection of these relational definitions. In terms of physical manipulation of the data, most relational databases are supported by a high level SQL (Structured Query Language) or SQL-like language to enable data definition, manipulation and querying within the relational schema rather than requiring separate languages for each of the tasks [172].

- ***Extended relational***: relational structures are suitable for simple data structures built from atomic types like integers, reals and character strings [121]. However, they are frequently inadequate for representing the structure of newer applications which require the ability to store complex or structured items [89]. These items include computer-aided design (CAD) drawings, bitmap images, multimedia sound and video files and large structured text documents for office automation and document management systems [338]. The extended relational approach in products such as *OpenIngres* enables support of these large and structured types as well as allowing programmers to defined their own special-purpose types [82]. Compound attributes may be modelled and representation of specialisation and generalisation is allowed. The extended approach also enables the creation of a temporal database thereby providing access to previous states of entities within the database. This persistence can be of immense benefit within SCM in terms of storing the component and configuration developmental histories, without the need to archive every piece of changed data.

- ***Object-oriented***: the object-oriented approach to database systems is intended to further advance the features of the relational approach [384]. In particular, better support is provided for the representation of large structured objects such as those found in CAD (Computer Aided Design), GIS (Geographic Information Systems) or document management systems. Object-oriented databases have at their foundation the concept of data and its functions encapsulated within a single object. In database terms the data encapsulated within an object is a tuple composed of other objects represented by their identifiers. Hence this has great possibilities within the ISCM process for representing configuration lists. Additionally, in common with the extended relational model, the object-oriented approach supports persistence and hence can keep control of permanent as well as temporary data. However, whilst object-oriented databases provide increased functionality, they require careful control with respect to schema evolution to ensure that any changes to the system preserve the structural and behavioural consistency of the database. In particular, it must be ensured that any changes made to a class are propagated fully through the inheritance lattice through which the concepts of specialisation and generalisation are supported [278].

- ***Parallel and distributed***: increasingly there is a move towards database models of a parallel, distributed [383] or federal nature [91]. Typically, modern information systems are based around a client-server model of computing, where the clients make requests of the server. In the past there has tended to be a centralised server, however new technology and the distributed nature of many businesses means there is a gradual move towards distributed architectures. Distributed databases may be homogeneous, that is, there are several sites all running their own applications on the same DBMS (Database Management System), or they may be heterogeneous where several existing databases, using different DBMSs, are linked into a single system. Federated database systems go a stage further and are a collection of independently managed, heterogeneous database systems that allow partial and controlled sharing of data without affecting existing applications. With the increasingly distributed nature of computing, the need to combine heterogeneous products with the application system and the need to enable distributed maintenance, these models are becoming more popular. This is particularly true with regard to the high storage needs of many of the new multimedia types and large document formats for which storage on one site of all data is not always physically possible. The need for improved access and retrieval of large structured information types such as sound and video has also meant that parallel architectures for databases, which can offer improved performance are also becoming popular [404].

Whilst there is an obvious need to store the logically represented ISCM information in a physical format, the mechanism to be adopted is not the primary concern of this thesis. Indeed, in terms of storage of the components themselves there is an argument for retaining the host system approach. This has been the approach taken within the scope of this work. However, in terms of representing the component descriptions and configuration lists, both a flat-file and extended-relational approach have been adopted, the reasons for which will be discussed in later Chapters.

3.5.2 Library Organisation

In addition to consideration of how to store and link individual pieces of configuration information, organisation of complete baseline versions must also be considered. The organisation and implementation of libraries may vary for different system environments and the particular tool being used. For example, some libraries may deal specifically with source code files, and others with object files, some may hold files or functions specific to a particular function or set of functions, and others might be organised to reflect the differing states of baselines or SCIs. Some systems distinguish between a program library and development library by considering a program library to be simply a store for compiled code and interface information, whilst a development library will also store the source code, analysis results etc. The *CLEMMA* [333] configuration librarian acts as a central repository for objects and information produced during a project, including source modules, test data, object code, specifications and manuals. Derivation histories, relationships, authorship, membership and access permissions are also stored in the library. DSEE [236] has the concept of 'object pools' which hold the binaries and other objects produced during the course of system building and associated build descriptions. Other systems operate on the basis of tiered levels of development, integration, system and official baseline software libraries with increasing levels of configuration control to channel the flow of completed software units to delivery. Softools *CCC* [356], for example, maintains this concept by having an ordered sequence of development, test, build, prototype and production accounts. Again, this area is not of primary interest within the scope of this thesis but it has been addressed within the *PISCES* M^4 system prototype through the use of versioned directories and *makefiles*.

3.6 Summary

This chapter has reviewed a number of development and maintenance process models. It is apparent that development-oriented process models do not adequately represent the maintenance process. Whilst this deficiency is now being addressed by a number of maintenance-oriented models, these models tend to approach maintenance from a high-level perspective and do not specifically address the activities required to model legacy systems. Given the high proportion of legacy systems in the 'real' world that need to be economically maintained, there is a pressing requirement for the development of a model that specifically addresses the problems of bringing legacy systems 'back under control'. This can be most effectively achieved through emphasis on the program comprehension activity during the maintenance process. In order to address program comprehension issues of legacy systems, the ISCM process will be defined in Chapter 4 and together with its associated model and method developed more fully throughout this thesis.

A software system is a complex product realised through the interaction of many heterogeneous components. These components in viable combination form the configuration of a software system. As the components of a system evolve for reasons of adaptation, correction and enhancement new configurations are generated such that a family of closely related but distinct systems form. It is the role of ISCM to control and uniquely identify this evolution of program families. Additionally, as the software discipline matures, systems are advancing such that the nature of the interactions between their component parts and the complexity of the interactions of these component parts with the system environment and third part tools is becoming increasingly complex. It is therefore no longer sufficient to limit modelling to the application components alone. For this reason as part of the ISCM process a set of component types or groups from which an application and its environment may be modelled will be identified and defined within Chapter 4.

This chapter has also reviewed a number of module interconnection languages (MILs) and related approaches currently in use for the modelling of software system architectures/configurations. An evaluation was made of the key features of each of these approaches and of their applicability to the ISCM process. As a result, a new language, the Inverse Configuration Description Language (ICDL), will be defined within Chapter 4 to enable the reclamation of legacy software system configurations and description of the resultant system models.

ISCM will be shown to have parallels in the activities of the traditional SCM activities. This is particularly so in the area of software manufacture, where a viable configuration has to be built by selecting compatible components, one per version-group. Although the focus of ISCM is on identifying an existent configuration rather than rebuilding a new one many of the same issues have to be considered and activities undertaken, and hence Chapter 4 will also explain how ISCM can be used a precursor to the build process.

It is also essential that the information reclaimed about a software system is stored in a format suitable for reporting and update. Advances have been made in the underlying mechanisms for storage of system configurations, their constituent components and associated information. These include extensions to the underlying file structure of the operating system, and the use of relational and customised databases. Additionally, research is active in the spectrum of areas from entity relationship through to object wrappers and mediators to fully object-oriented approaches to object storage [82]. Storage of the configuration 'objects' and rules within the ISCM process is the primary role of the Extensible System Information Base (ESIB).

Chapter 4

Inverse Software Configuration Management

And as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name

Shakespeare A Midsummer's Night's Dream, Act V, Scene I

4.1 Introduction

This chapter describes the key characteristics of the **Inverse Software Configuration Management (ISCM) process** and development of its resultant **maintenance process model**. ISCM is defined within this chapter as *the process of bringing existing (operational or legacy) software systems back under configuration control.* In order to realise control of a legacy system a theoretical understanding of how software system architectures can be modelled is required, coupled with a practical method for identifying and locating the component parts of the model from within a 'live' application. This chapter consequently defines the component types of a system, recognised within the ISCM process, as being essential to the effective modelling of software system architectures. ISCM involves modelling a given application system at a number of architectural levels. This chapter therefore also details the roles and relationships of each component type in terms of a number of abstract system configurations, for example at baseline, version and environmental levels.

Following identification, these components and their relationships need to be formally defined and documented within the context of a software configuration and corresponding system model. There is therefore a need to syntactically describe the configurations resulting from component combination. With this in mind and with respect to the review of configuration description languages carried out in Chapter 3, this chapter also develops the **Inverse Configuration Description Language (ICDL)** based on a balance between current best practice and applicability to the ISCM process.

The ICDL provides a convenient 'machine-oriented' file format which can be subsequently exploited during the semi-automation of the ISCM process via the **PISCES M**^{4} tool series. However, the 'human-aspects' of the process must also be considered, these human-aspects are facilitated by representing the system according to a ***Proforma Increasing Complexity Series*** (PICS). Within the PICS a series of proformas of similar structure but increasing application complexity provide the mechanism for the collection and collation of information for underpinning the knowledge base and describing software system architectures. The PICS templates in their role as a natural language mechanism for software configuration reclamation and description are therefore developed within this chapter. Mention is also made of how an inverse perspective to the traditional system building process can be used to incrementally populate the PICS in order to reclaim a software system configuration.

4.2 Inverse Software Configuration Management

Whilst numerous process models now describe the maintenance process, none of them address explicitly the problems of bringing a legacy system back under configuration control. This thesis proposes the process of Inverse Software Configuration Management (ISCM) as a mechanism to achieve a controlled and repeatable method of cost effectively reclaiming 'out of control' legacy systems. Thus, there is a need for the reasons discussed in section 3.2.3 to enhance and extend current maintenance process models as well as embody the best practices encompassed within them. This augmentation must encompass the detailed configuration management activities required in order to re-establish control of a system. This section develops the ISCM process model and discusses the key underpinning features embodied within it. The key terms associated with the ISCM process are defined in Table C4-1.

Term	Definition
ISCM Process	The overall process and its definition.
ISCM Maintenance Process Model	The conceptual stages/activities necessary to effect the ISCM process, and its relationship to the global software lifecycle.
Process Architecture	The modelling tool to represent the process model at varying levels of abstractions.
PISCES Method	The physical realisation of each of the above stages/activities through a defined sequence of steps and templates
PISCES M^4 System	The meta-CASE environment developed to semi-automate the ISCM model & PISCES method

Table C4-1 ISCM process model terms

4.2.1 Process Model Representation

Process models may be represented at a number of different levels and for a number of different client groups, for example management, customers, developers and even lawyers involved in litigation regarding software products and development [256, 278]. Additionally, even within the different client groups it may be advantageous to view a system from a number of different architectural perspectives. However, the majority of development and maintenance models described in Chapter 3 may be considered to be high-level, task-oriented models approached from a single viewpoint. In order to more accurately define the ISCM process such that it can provide useful results for a wide-ranging customer base including managers, customers and maintainers, there is the need to address the process model from a number of levels of abstraction. Additionally, in order to ensure consistency and traceability of the ISCM process there is the need to provide not only a high-level guide to the tasks that need to be conducted, but also detailed guidance for how a particular task will be achieved. Consideration will therefore be given to three levels of process abstraction: policies (overall process - managers/customers); procedures (high-level tasks - managers/customers) and work instructions (detailed activities - maintainers).

The modelling approach will be supported by software process architecture diagrams as the means of describing the processes, their associated activities, and the corresponding relationships and dependencies that exist between processes. Levels zero, one and two of the Inverse Software Configuration Management (ISCM) maintenance process model will be described in section 4.3. Level three, which is synonymous with the PISCES (Proforma Identification Scheme for Configurations of Existing Systems) method, will be addressed in depth within Chapter 6.

4.2.2 The Inverse Software Configuration Management Process

Inverse Software Configuration Management (ISCM) may be defined as the ***process of bringing operational (existing or legacy) systems back under configuration control*** [264, 266]. Like traditional Software Configuration Management (SCM), ISCM encompasses the four basic elements of [43]:

- ***Identification*** - defining and uniquely identifying the baselines and corresponding components of a system and any changes made to the components of the baselines.
- ***Control*** - managing through defined procedures any changes made to the components and the baselines of a system.
- ***Status Accounting*** - providing an administrative history of how the system has evolved and its current status.
- ***Audit*** - determining that defined baselines meet their requirements and that the control, identification and accounting procedures have being adhered to correctly.

The major difference between maintenance-oriented ISCM and development-oriented SCM lies in the far more comprehensive treatment of the *configuration identification* phase of the process. Within the ISCM framework this is termed **Inverse Software Configuration Identification** (ISCI). The increased importance of this phase for existing or legacy systems is due to the extra program comprehension burden attributed to the reasons discussed in Chapter 2. That is the:

- Increased complexity of a system through its evolution over time.
- Progressive increase in the number of versions of a system.
- Probability of lost or poor associated documentation.
- Personnel changes leading to loss of undocumented system knowledge.
- Monolithic nature of software developed prior to modularisation techniques.
- General lack of experience regarding the development environment of the system.
- Gradual degradation of system structure and introduction of 'ripple-effect' errors.
- Tendency for heterogeneous implementation techniques to become used over time.

Additionally over the lifetime of a system the executing application can become cluttered and masked through the proliferation of new versions, 'alien' files from other applications and updated environmental components etc. [4]. The application may have also become fragmented if a strictly controlled file and directory structure has not be maintained, leading to possible misplacement or even irretrievable loss of application components. During the ISCI phase there is the requirement to draw together the relevant system component information and their associated dependencies and to censure that which is no longer or has never been required.

4.2.3 Inverse Software Configuration Identification

In order to address the complexities of legacy systems, Inverse Software Configuration Identification (ISCI) encompasses a number of related activities. These include component identification, component relationship understanding, component location mapping, incremental documentation and multimedia representation of each of these information sources. These activities will be outlined in this section and will be described in more detail throughout the remainder of this chapter and the rest of the thesis.

4.2.3.1 Component Identification & Relationship Comprehension

The key objective of the ISCI phase is to enable the system being maintained to be understood in order that changes can be made cost-effectively and without introducing any further errors to the system. With regard to this, the ultimate aim of the process is to recover the operational system configuration plus develop the means to assess the potential impact of any changes on the system. In order to perform effective program and system comprehension it has been shown that a wide range of informal, semi-formal and formal information types are useful [225, 234]. The ISCI activity thus considers it essential to gain understanding of a system and its components at several levels of abstraction and related to several areas. These are: the application itself; the version groups of the application; the problem domain of the application; the cause-effect relation of the change to other parts of the system; the relationship of the application to its environment and the high-level business oriented design support facilities.

As systems become more complex in terms of the number and diversity of components so do the relationships between the components and the way they interact with each other and with the operational environment. Consequently, it is not sufficient to only extend identification to the application configuration components as there is a need to specify, identify and understand [265] the:

- ***Intra-specific relationships*** - those existing between the components within an application at a source code level (the traditional view of program comprehension).

- ***Inter-specific relationships*** - those existing between the components of an application and the other identified components types (the extended ISCM (or system) view of program comprehension).

For effective legacy software system comprehension the component types shown in Figure C4-1 have been identified as being important in order to gain an overall understanding of the system. The products of the ISCI activity are a number of configuration descriptions at varying levels of abstraction, namely the baseline (operational system) configuration; the version group (program family) configuration; and the environmental (global linkage) configuration. Detailed descriptions of each component grouping, the resultant configurations and their role in the program comprehension process are given in section 4.4

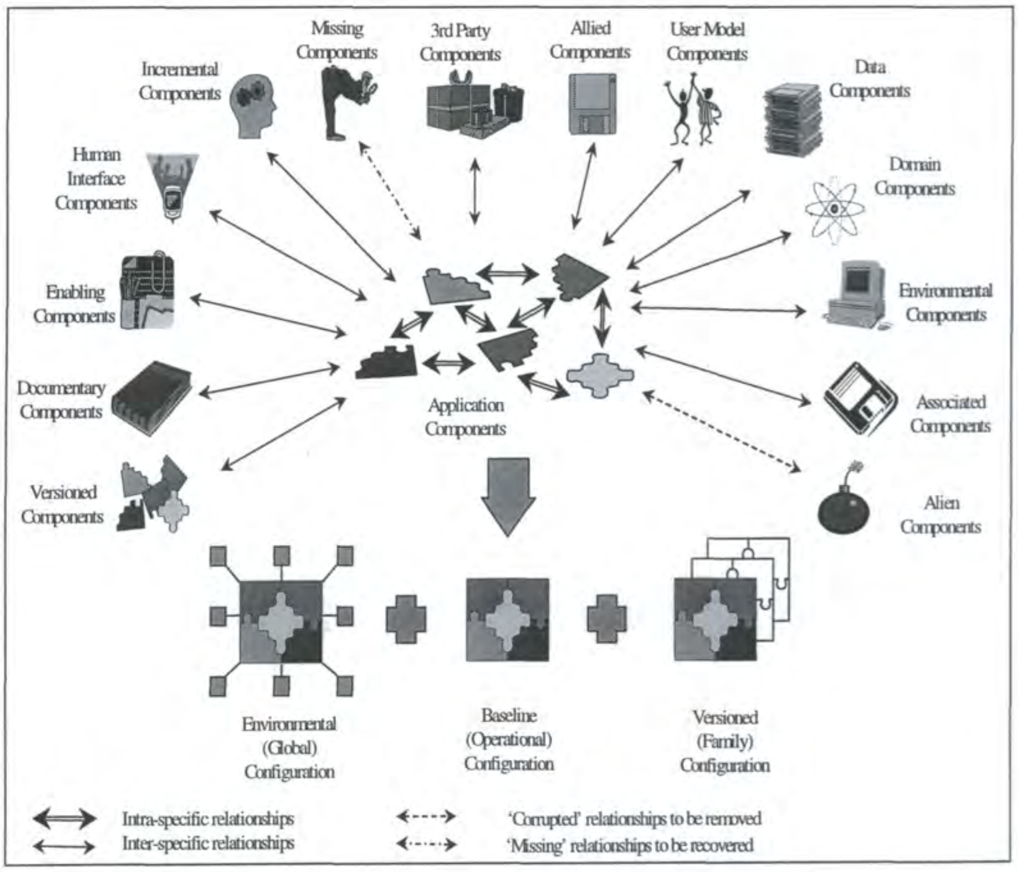


Figure C4-1 ISCI component groupings

4.2.3.2 Component Location Mapping

Whilst the primary emphasis of ISCI is the identification of system configurations and their relationships, another important element of the ISCI process is that of component location mapping. This involves determining the location of each of the identified configuration components within the file structure imposed by the operational environment. This process results in the production of configuration location maps or paths, which aid component retrieval for maintenance purposes. This is particularly important if an ordered project environment has not been adhered to or has become cluttered over time. Mapping of the component locations facilitates the restructuring of the application environment such that the storage and retrieval of the application file types can become more controlled and more rapid during future product changes. This is especially useful when it comes to identifying the missing and 'alien' or redundant component types.

4.2.3.3 Incremental Documentation

Associated with the ISCI process is the concept of incremental documentation [148, 265]. This enables information obtained during the maintenance process to be documented with the minimum of extra effort or overhead. Additionally as it can be carried out during maintenance it is likely to be more accurate and complete than if documented post-maintenance. This increases the likelihood of the documentation being kept concurrent with the state of the application. Incremental documentation techniques also ensure that only the effected parts of the system are documented resulting in significant cost benefits (in accordance with the 80:20 maintenance rule, that is, eighty percent of time spent on maintenance is spent on twenty percent of the code). Another advantage of this approach to documentation is that it enables the knowledge of one maintainer to be preserved for future maintainers, thereby enabling the program comprehension process to become progressively less time consuming. Such an approach also enables a 'maintenance oracle' [268] to be progressively defined.

4.2.3.4 Multimedia Representation

Incremental documentation of the knowledge gained through the program comprehension process means that the same degree of investigative work required to understand the system does not need to be repeated for the same area of code each time a change is required. However, in order to gain the maximum benefit from this approach it is also important to be able to represent any reclaimed knowledge in a way that can be productively used by the maintainers. Thus, the ISCM method represents the information gleaned during the ISCI activity as a mix of textual and graphical output combined through hypertext links [264]. Recent research has extended this representation to take advantage of the rich multimedia attributes of video, audio and animation [268]. Graphical output from this activity may be at each of the three levels of configuration abstraction, that is, baseline (operational) dependency networks, program family diagrams and environmental (global) linkage plans, together with the file location maps of the identified components.

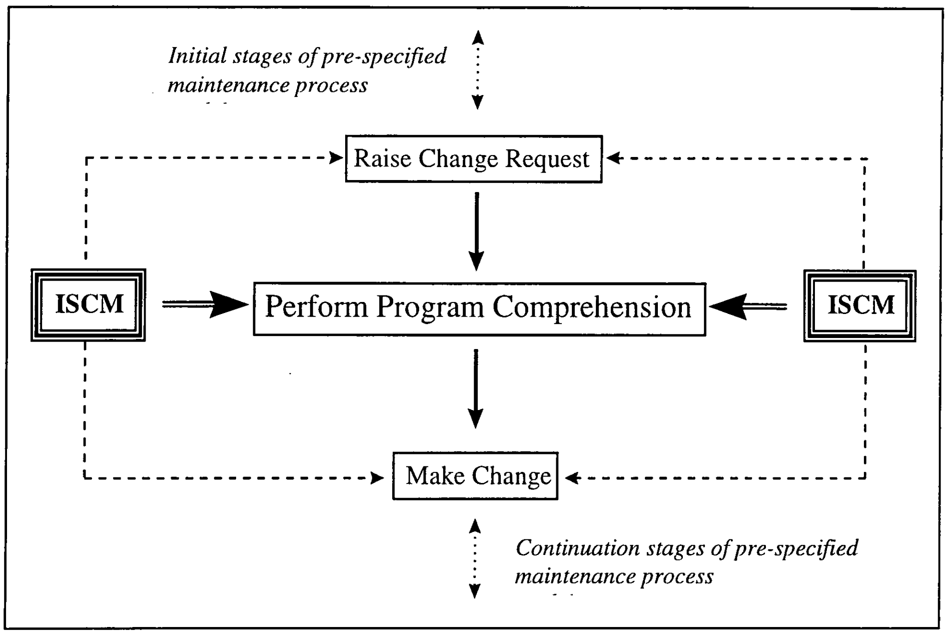
The activity also generates higher level abstractions of cognitive and traditionally documented information that have particular relevance to the program comprehension process. The mechanism for production of these output types will be discussed in detail in Chapters 5 and 6.

4.2.3.5 Modelling Techniques for System Abstractions

Whilst the configuration diagrams and file location maps provide 'snapshots' of the system at a particular point in time, it is also necessary to represent more persistently and rigorously the configurations of the application and associated environment. In order to take into account the needs of the different customer groups it is considered necessary to represent the reclaimed system configuration information at varying levels of abstraction. The information progressively recovered with regard to a system is stored in an Extensible System Information Base (ESIB) and may be modelled at differing abstractions through the use of a series of natural language proformas. Additionally these natural language proformas act as the collection mechanism for information about an application with which to populate the ESIB.

4.3 The ISCM Process Model and Architecture

As indicated in the previous chapter there are a number of different process models and approaches to maintenance currently in use. Whether or not their primary emphasis is on economy of changes, quality control of the process or searching for component reuse etc., it is evident that all models recognise the need to analyse the change, understand the affected part of the system and assess the potential impact of any changes on the current system equilibrium. With this in mind, it is the intention of the ISCM process model to elaborate the comprehension activities of current process models rather than to consider itself as a complete maintenance model to replace those currently in use. In this way the ISCM process can be incorporated into each of the currently defined process models thus capitalising on the experience of these models whilst adding value to the maintenance process. Additionally this mechanism causes minimal disruption to organisations who have already adopted a specific maintenance approach and satisfies the requirements for flexibility within process models [156] whilst still maintaining a high degree of rigour. The ISCM process model therefore fits into the global framework as shown in Figure C4-2.



Initial stages of pre-specified maintenance process
Raise Change Request
ISCM
Perform Program Comprehension
ISCM
Make Change
Continuation stages of pre-specified maintenance process

Figure C4-2 The role of ISCM within the global process model framework

Whilst the low-level activities of the ISCM process model will be detailed in Chapter 6, this section presents a high-level overview of each phase of the ISCM approach. Underpinning the model is the reconstruction of the information required about a system in order to perform effective program comprehension prior to making a change, plus the subsequent storage, retrieval and representation of this information for maintenance purposes. The ISCM process model is described in terms of the pre- and post-conditions for each activity; the inputs required to effect each activity output; and any feedback into previous phases of the model.

4.3.1 Overall Process Model

At the highest level of representation the ISCM process model is embedded within an existing process model framework. This existing framework guides the maintenance process up to the point of raising a change request, which may be before or after cost-benefit approval of a change depending on the model in place. Alternatively, if a maintenance model is not in place, the change request and tracking procedures encompassed within the ISCM approach for such an eventuality can be evoked. The ISCI phase of the ISCM process is then initiated to analyse the change, understand the affected part of the system and assess the potential impact of the proposed change on the system. After the ISCM process activities have been conducted and the configuration information reclaimed in order that the system may be brought back under control, the existing framework for the maintenance process is resumed in order to carry the change process through to completion. Subsequent activities within the framework must include the application of all the activities of the SCM (and ISCM) process, namely software configuration identification, configuration control, status accounting and audit. The high-level representation of the model is shown in Figure C4-3.

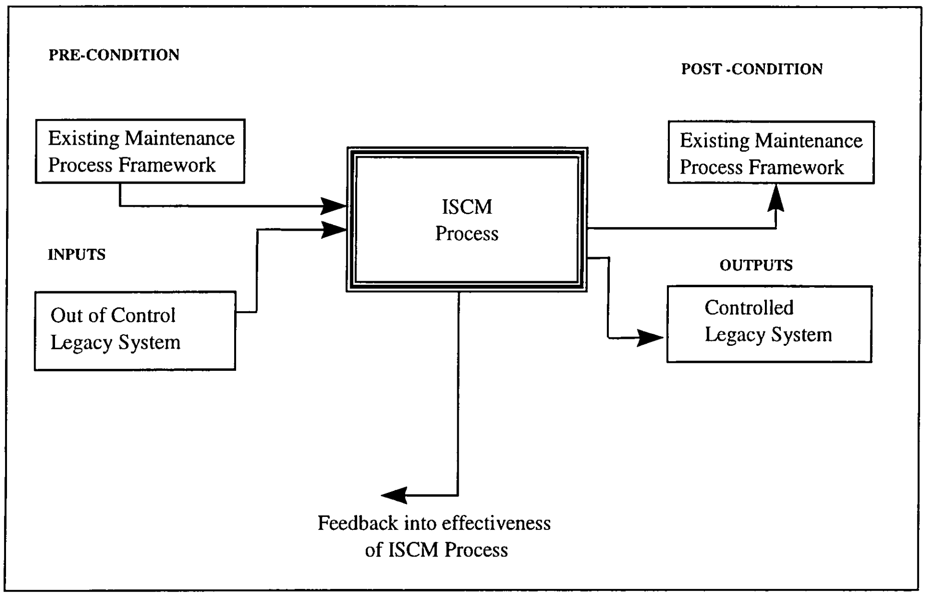

PRE-CONDITION
POST -CONDITION
Existing Maintenance Process Framework
Existing Maintenance Process Framework
ISCM Process
INPUTS
OUTPUTS
Out of Control Legacy System
Controlled Legacy System
Feedback into effectiveness of ISCM Process

Figure C4-3 Level-0 : overall (policy) level ISCM process framework

The activities that combine to form the overall ISCM process are shown in Figure C4-4. Each of these activities is further decomposed in the Level-2 process architecture diagrams and will be expanded fully in the following chapters of the thesis.

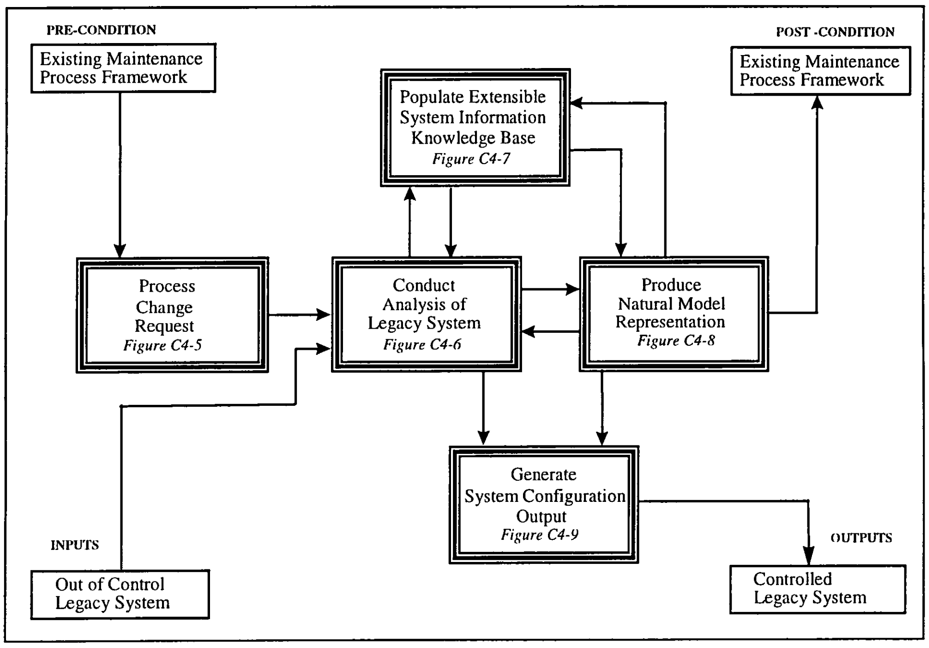

PRE-CONDITION
POST -CONDITION
Existing Maintenance Process Framework
Existing Maintenance Process Framework
Populate Extensible System Information Knowledge Base Figure C4-7
Process Change Request Figure C4-5
Conduct Analysis of Legacy System Figure C4-6
Produce Natural Model Representation Figure C4-8
Generate System Configuration Output Figure C4-9
INPUTS
OUTPUTS
Out of Control Legacy System
Controlled Legacy System

Figure C4-4 Level-1: high (procedure) level ISCM process model

4.3.2 Process Change Request

Maintenance, whether in association with correction, perfection, adaptation or prevention should always occur in response to a formally raised change request. As mentioned previously, the responsibility for raising this request may reside within the existing maintenance framework, or may be incorporated as part of the ISCM process itself. However, whatever the origin of the change, it is important that the reason for it is fully documented. The change must also be associated with a particular software component and should be given a unique identification number so that its status can be tracked throughout the maintenance process. There may also be a change log for the component already in existence in which case this also becomes an input to the process. The aim of the 'process change request' activity is thus to formally document and identify the change and any relevant information prior to analysis of the legacy system in order to understand and assess the impact of the proposed change. If during the change request process, a particular documentary item needs to be created, a blank proforma of the required type is raised, which when populated or semi-populated with information then assumes the specific role for which it is intended. The change request activity is summarised in Figure C4-5.

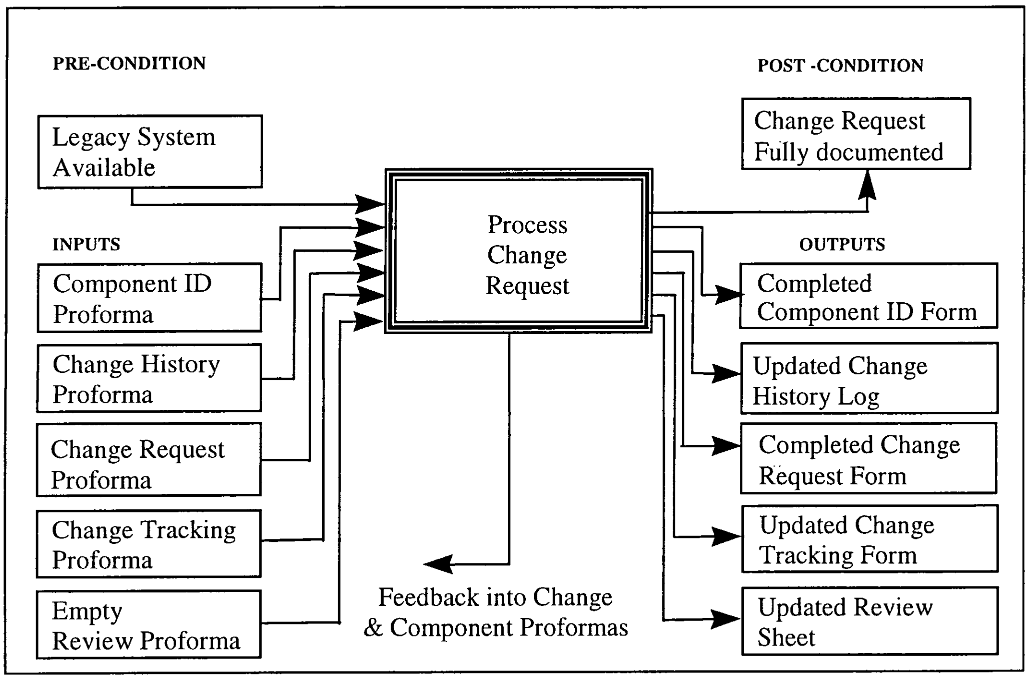
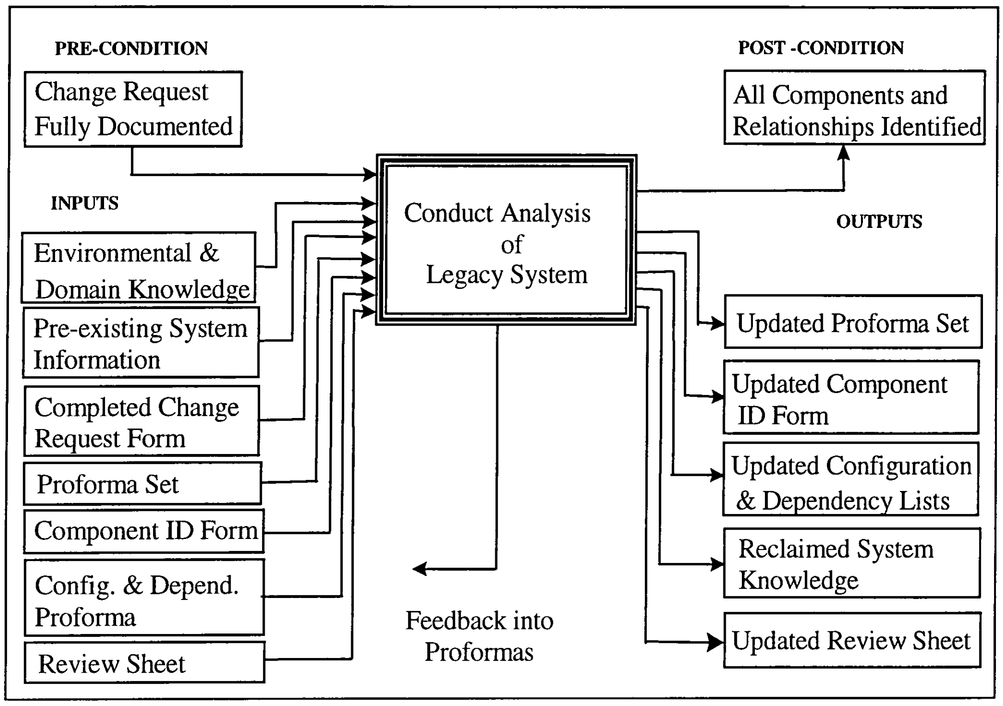


Figure C4-5 Level-2 : process level - ISCM change request process

4.3.3 Conduct Analysis of Legacy System

The key aim of ISCM and the Inverse Software Configuration Identification (ISCI) phase in particular is to identify that system configuration existing at a particular point in time as well as its interactions with the environment (Figure C4-6).



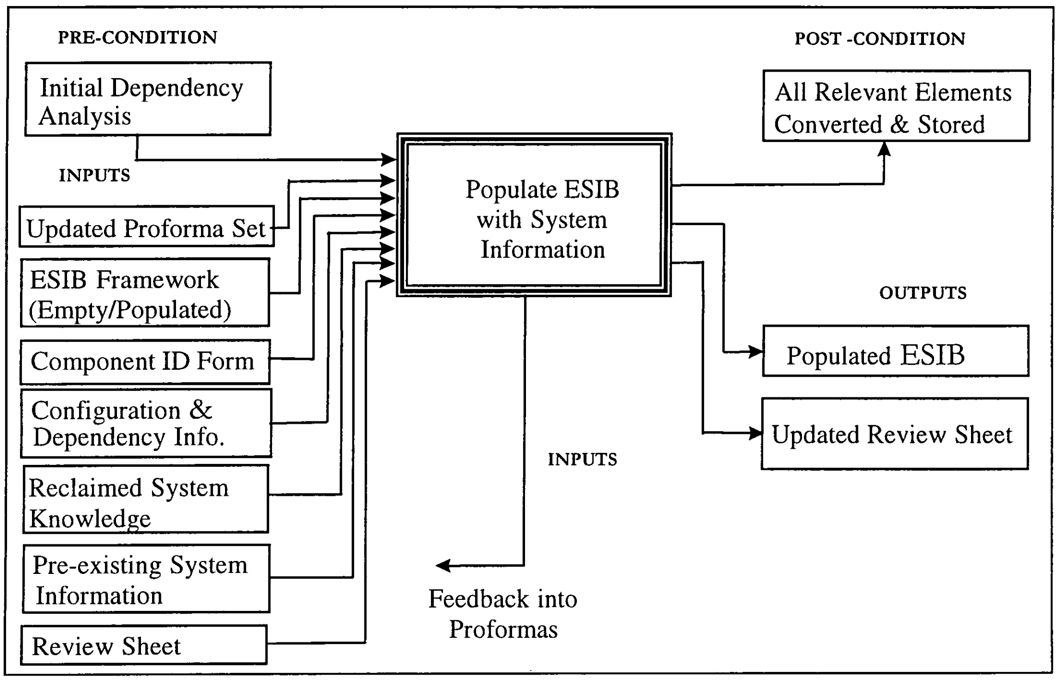
PRE-CONDITION
Change Request Fully Documented
INPUTS
Environmental & Domain Knowledge
Pre-existing System Information
Completed Change Request Form
Proforma Set
Component ID Form
Config. & Depend. Proforma
Review Sheet
Conduct Analysis of Legacy System
Feedback into Proformas
POST -CONDITION
All Components and Relationships Identified
OUTPUTS
Updated Proforma Set
Updated Component ID Form
Updated Configuration & Dependency Lists
Reclaimed System Knowledge
Updated Review Sheet

Figure C4-6 Level-2: process level - ISCM conduct analysis of legacy system process

There is therefore the need to carry out a comprehensive analysis of all the component types alluded to earlier in Figure C4-1. The starting point for this activity is any pre-existing knowledge or documentation about the system and its environment. This includes which, if any, information extraction tools are available on the system host environment. The comprehension process is aided by the developed proforma series of templates, which enables information about a system to be progressively built-up and documented until a complete system configuration has been established. All information extracted about a system is stored in the Extensible System Information Base (ESIB). The ESIB becomes increasingly populated at a *specific* level, as a particular system is maintained, and at a *general* level as different systems are maintained. This progressive build-up of information means that initially the proformas will be empty of information but will act as guides to the reclamation process. However, as more information becomes held in the ESIB partially or fully completed proformas become the starting point for the comprehension process.

4.3.4 Populate Extensible System Information Base

Once the information about a system has been collected it is important to retain this information and any new information recovered about a system. The role of the Extensible System Information Base (ESIB) is therefore to serve as a repository for this information and the nature of any relationships between information types. As the EISB becomes progressively more populated with information it may also be used as a mechanism for semi-automated and eventually 'intelligent' program comprehension of a system, thereby reducing the manual effort required by the maintainer. A review list forms part of the audit requirements of the ISCM process and ensures that the correct process has been adhered to and that all component groups and relationships have been considered.



PRE-CONDITION
Initial Dependency Analysis
INPUTS
Updated Proforma Set
ESIB Framework (Empty/Populated)
Component ID Form
Configuration & Dependency Info.
Reclaimed System Knowledge
Pre-existing System Information
Review Sheet
Populate ESIB with System Information
INPUTS
Feedback into Proformas
POST -CONDITION
All Relevant Elements Converted & Stored
OUTPUTS
Populated ESIB
Updated Review Sheet

Figure C4-7 Level-2 : process level - ISCM populate ESIB with system information process

4.3.5 Produce Natural (Proforma) Representation of ISCM Model

Whilst the proformas provide the driving force behind the collection of information with which to populate the ESIB, they also serve the purpose of deriving the system configurations at varying levels of abstraction. Information pertaining to each of the component groups stored in the ESIB is represented for a particular application as part of a Proforma Increasing Complexity Series (PICS). Each PICS represents a software system architecture at three levels of abstraction namely, generic, tailored and specific. The structure of each of these abstractions will be described in detail in Section 4.6, but the proforma creation process is represented in Figure C4-8.

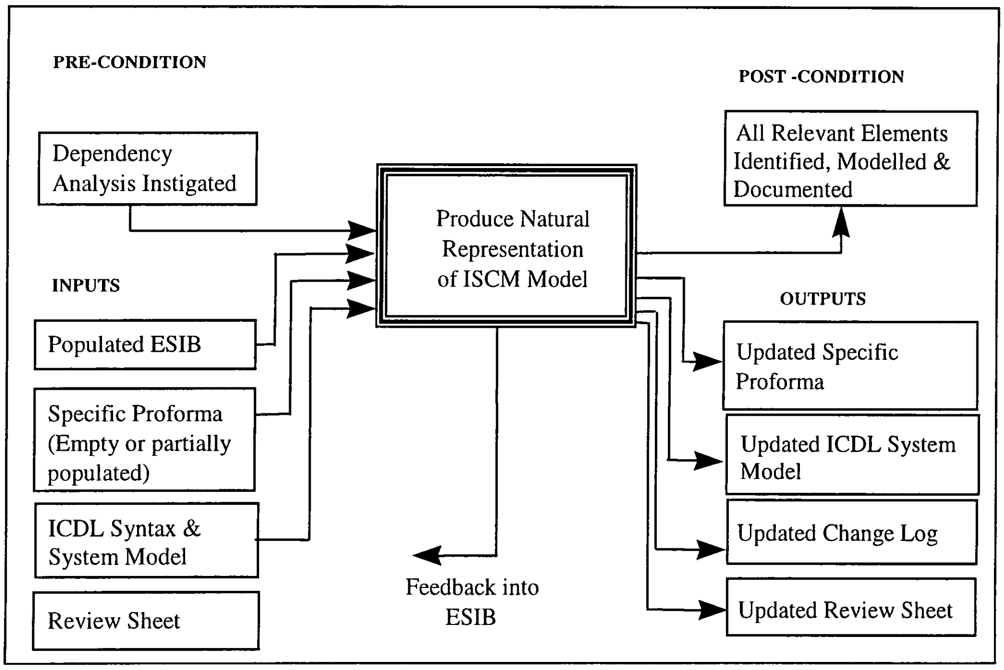

PRE-CONDITION
Dependency Analysis Instigated
INPUTS
Populated ESIB
Specific Proforma (Empty or partially populated)
ICDL Syntax & System Model
Review Sheet
Produce Natural Representation of ISCM Model
Feedback into ESIB
POST -CONDITION
All Relevant Elements Identified, Modelled & Documented
OUTPUTS
Updated Specific Proforma
Updated ICDL System Model
Updated Change Log
Updated Review Sheet

Figure C4-8 Level-2 : process level - ISCM production of natural language proforma process

Systematic processing of the different levels of system abstraction enables information to be cumulatively added to the proformas thereby increasing their level of complexity and decreasing their level of abstraction. Ultimately a '*genetic fingerprint*' of a particular application system as at particular point in time is achieved which describes the baseline, version group and environmental linkage configurations of the system.

The proformas therefore serve two purposes, as a guide for reclaiming information and as a natural language representation of the information held in the ESIB. The completed proformas can also be stored and used as a history of how a complete system configuration has evolved over time in a similar fashion to the derivation produced for a single software configuration component.

In order to assist in automating the ISCM process, an Inverse Configuration Description Language (ICDL) has also been defined and is used concurrently to syntactically document the system configuration information represented by the PICS series in a more machine-readable format. The ICDL is defined and described in section 4.5.

4.3.6 Generate System Configuration Output

In order that the system configurations provide useful information, it is important that the information they contain is clearly presented to the maintainers. The aim of this activity is therefore to present in textual, graphical or multimedia format the system information extracted during the program comprehension process. The specific proforma of the PICS and the ICDL provide the input to the activity resulting in output such as baseline configurations, program family diagrams, environmental linkage plans, file location maps etc. Figure C4-9 shows just a small selection of the types of output that can be produced. This output is discussed further in Chapters 5 and 6.

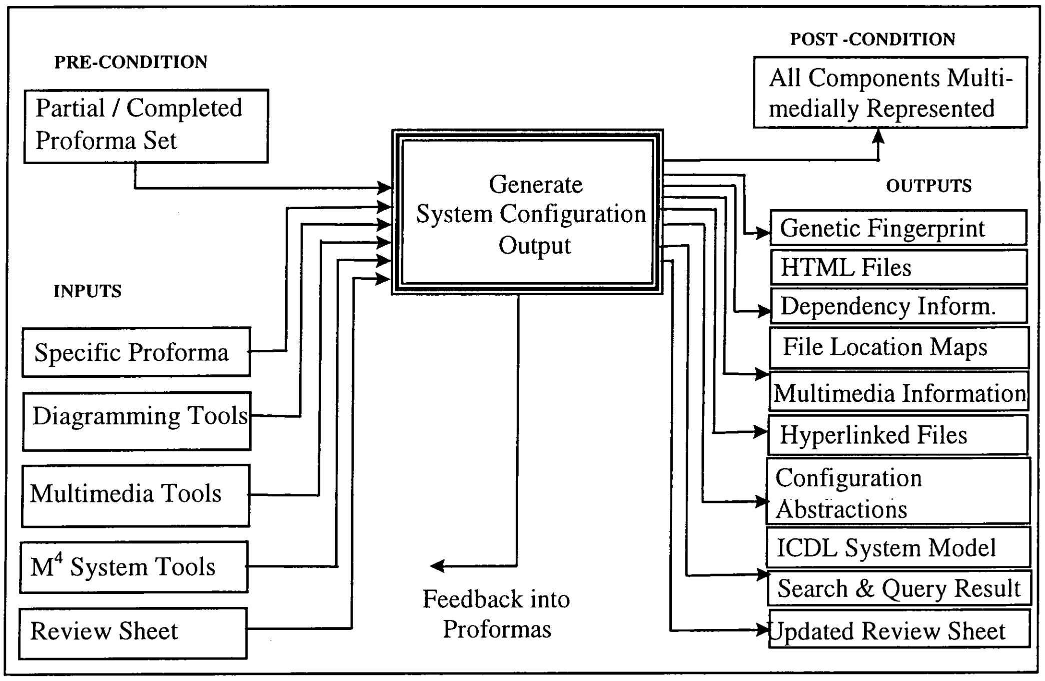


Figure C4-9 Level-2 : process level - ISCM generation of system configuration output process

4.4 System Building Blocks within the ISCM Process

At the centre of the ISCM process described in the previous section is the ability to model the new generation of software systems in terms of their component parts and the relationships existing between these parts. The thesis will address the issues of this 'new breed' of software system by defining ways of building up configurations/architectures at varying levels of abstraction. This work builds to some extent upon the generic and baseline configurations of Tichy [376] in terms of adding levels to the set of configurations in the form of family and environmental configurations. It also encompasses many of the ideas of Sacchi [342] who defined very specifically and in detail what sort of information should be maintained about a system component. It must also be remembered that this information has to be recovered from legacy systems and hence there must be a mechanism for the progressive population of the information base and records about each component. The approach to the work although developed independently, also has some similarities to the modelling techniques employed by Sommerville [360], although there are also other distinctions that can be made and which will be discussed later in the thesis. Additionally, the identified problems arising through having to maintain configurations lists are addressed via one of the original elements of the ISCM process. The mechanism for maintaining configuration lists will be discussed in more detail in Chapter 5, but in essence configuration lists may be maintained efficiently through the use of stored references to other documents. This is now relatively easy to implement through the use of hypertext and hypermedia links.

The following sections of this chapter describe a number of mechanisms for modelling software configurations/architectures. However, as indicated in Chapter 3 and commensurate with a developing field of research there are the discrepancies often associated with 'settling' terminology and in this case the possible ambiguity over what is meant by the term 'software architecture' and its related concepts. Hence, for the purpose of this thesis the definitions given in table C4-1 have been assumed within the context of the ISCM process.

Software (System) Model	This is the overall abstract representation of a software system. It is essentially a conceptual model defining the entire set of components from which a software system may be composed and the possible relationships existing between the component types.
Software System Architecture	This is a more refined representation of a software system. It is still conceptual and logical in nature but deals with a lower level of granularity in terms of describing particular abstract representations of a system. In essence, this is the overall abstract and generic representation of a software system. It is essentially a conceptual model defining the entire set of components from which a software system may be composed, the attribute types of these components and the possible relationships existing between the component types and their interaction with the rest of the system
Software Configuration Component	This is the fundamental building block of an application system. It is at the level of an individual physical item within a system (i.e. not a procedure or function within a program or module)
Software Configuration	This is the physical combination of components that together form a viable system. In essence it is the physical realisation of a particular software architecture. The combination may be at varying levels of abstraction and in particular at baseline, family or environmental. In essence this is the physical instantiation of the software architecture with component names, version information, and attribute values such that the viable software configuration is described.
Software Architectural Abstraction	A model of the software architecture from a particular viewpoint or defined level of granularity.
Software Configuration Abstraction	A model of the software configuration from a particular viewpoint or defined level of granularity.
Genetic Fingerprint	The lowest level and unique description of a software system configuration at a particular point in time.
Attribute	This is a property of a particular component(s)
Location	This is the physical storage position of a particular software configuration component
Relationship	This is the nature of the bond existing between two components within a group (intra-application relationships) or between components in different groups (inter-application relationships)
Configuration History	The time-related series of generic fingerprints which progressively record or reclaim the evolutionary characteristics of a particular software system.

Table C4-2 Definitions of ISCM modelling terms

4.4.1 System Layers

As part of the ISCM process and in relation to the above discussion regarding what constitutes a software system architecture and the types of components which must be taken into account when developing or maintaining a system, a number of component groupings and layers have been identified within the scope of this thesis and as such will be encompassed within the ISCM model.

The component parts of a system may be best identified and described by considering a layered approach to software system developments. In this way it is possible to deal with the interaction of the system as a whole, which is a key issue in connection with both green field and legacy software systems.

A system configuration may thus be considered within this framework as a number of layers with which the application system will interact during its operation. The layered model is shown in figure C4-10 and may be represented by the component types described in the following section.

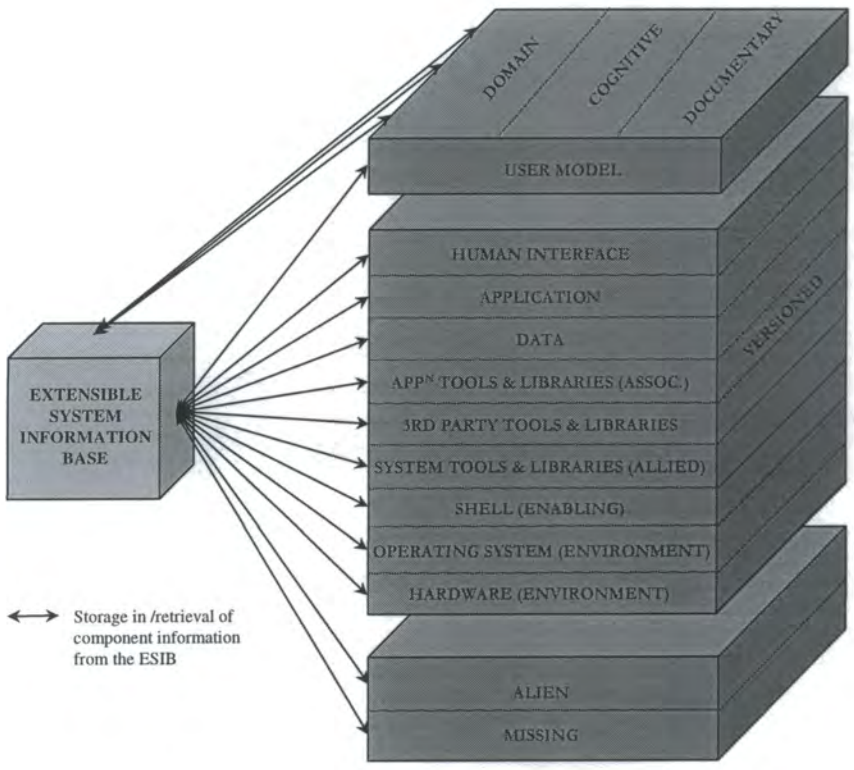

DOMAIN
COGNITIVE
DOCUMENTARY
USER MODEL
HUMAN INTERFACE
APPLICATION
DATA
APP^N TOOLS & LIBRARIES (ASSOC.)
3RD PARTY TOOLS & LIBRARIES
SYSTEM TOOLS & LIBRARIES (ALLIED)
SHELL (ENABLING)
OPERATING SYSTEM (ENVIRONMENT)
HARDWARE (ENVIRONMENT)
VERSIONED
ALIEN
MISSING
EXTENSIBLE SYSTEM INFORMATION BASE
Storage in /retrieval of component information from the ESIB

Figure C4-10 ISCM layered component model

4.4.2 Component Types

By taking the layered approach and considering the advances made in software systems development over the past few years and described in Chapter 3, the following types of component groups can be identified:

4.4.2.1 Application components *(Cg_{app})*

These are the central components of the application encompassing software configuration items (SCIs) such as the source code modules, object code, and dedicated libraries that comprise the actual running version of the system. They constitute what are considered to be the 'traditional' components of a software system and as such they form the central core of the baseline configuration of the system.

4.4.2.2 Versioned components (Cg_{ver})

These components are the variants and revisions of the system that result from changing components in the system. Collectively these form a version group at the level of a particular component and a program family at the level of a complete application system configuration. Components of all types may be versioned and the different versions must be tracked if viable configurations are to be created and maintained.

4.4.2.3 Domain components (Cg_{dom})

These components concern documented or cognitive knowledge regarding the domain environment in which the application is designed to operate. Such information assists in the understanding of the system context and may impact on the approach etc. required during the maintenance process. This knowledge may exist through original documentation or may have to be post-documented after knowledge elicitation from the users, developers and maintainers of the system.

4.4.2.4 Incremental components (Cg_{cog})

These components encompass the developers and maintainers knowledge regarding the functionality, structure and dependencies of the system that has been incrementally documented through the ongoing process of program comprehension. Hence these components primarily contain reclaimed knowledge and understanding of a software system configuration.

4.4.2.5 Documentary components (Cg_{doc})

These components represent the 'formally' produced and generally traditional documentation that supports the development of an application. These components include documents such as specifications, design documents, test plans etc.

4.4.2.6 User model components (Cg_{user})

The user would normally be considered to be 'outside' the modelling scope of an application. However, increasingly the user plays an important role in the design, development and maintenance of a software system regardless of whether this is in a proactive or reactive capacity. The ISCM model therefore considers that the user should be considered to be a component of the software system. Indeed the user can also be considered to be a versioned component as particular users may require specific settings for, or tailoring of applications.

4.4.2.7 Human interface components (Cg_{int})

Increasingly applications are making use of user interface languages or development tools to create a graphical user interface layer to the application. Whilst the main application may be written in the same language as the interface, for example Visual Basic or Delphi, it may well be that the underlying application has a completely different structure and has the GUI layer lying 'on top of' the application. This component group contains the interface components, some of which have been programmed from first principles, others of which are reused from a library of interface components and widgets.

4.4.2.8 Data components (Cg_{data})

These are the data file components, such as the database or spreadsheet data etc. that are consumed and produced as a result of running the application components. These components may or may not be part of the baseline configuration of the system, depending on whether or not data is stored and accessed or whether it is used at run-time, then discarded. If data is stored and accessed then it forms part of the secondary baseline level.

4.4.2.9 Associated components (Cg_{assoc})

These are components such as the application language libraries and reusable components that are associated with the application under development. Thus although they are not developed and unit tested as such, these components must still be maintained as part of the core application if a viable configuration is to be evident.

4.4.2.10 Allied components (Cg_{alli})

These form a group similar in function to the associated components. However, they differ in terms of ownership of the libraries etc. For example, they include the general system libraries and other operating system components that are required by the application, but which are not actually maintained as part of it.

4.4.2.11 Environmental components (Cg_{env})

These are components such as compilers, linkers, editors, tools and hardware that make up the operational environment of the system and as such they may impact heavily on the portability of the application system from one environmental upgrade to the next. They differ from the allied components in the nature of their function - they are involved with the creation of the program, rather than the allied and associated components whose library software becomes an integral part of the application. There may therefore be a distinction between enabling components (environmental) and inclusive components (allied and associated).

4.4.2.12 Third-party components (Cg_{trd})

These are the components that belong outside the application itself but with which it interacts. They may include database software, other tools with which OLE links have been made, or other proprietary software which is then customised by the user for their own purposes.

4.4.2.13 Enabling components (Cg_{enab})

These include the components that do not form part of either the core or secondary software system but which are tools to help with the control or program comprehension of the software application. Components of this type include version control systems and tools such as awk and grep which can provide definitive information about the state of the software and its dependencies. As such it is important to be able to model these components along with the software system in order that the host environment can be exploited to the full in terms of understanding the application system, particularly in relation to legacy software systems.

4.4.2.14 Alien (redundant) components (Cg_{alien})

These are components that are never referenced by the application system. They may be from another 'alien' application. Alternatively, they may be 'rogue' development versions which have been abandoned in the workspace as intermediate or non-implemented changes and which were never intended to be integrated into the actual executing version of the application system. These components are unique from all other components in that they have no part to play in the system configuration and indeed are detrimental to the comprehension of a system configuration through the ambiguity and uncertainly they cause.

4.4.2.15 Missing components (Cg_{miss})

These are components that are referenced by the system but which have been lost, destroyed or misplaced. Their loss may prevent the entire system being rebuilt, particularly if the components are only rarely referenced e.g. a module to convert the date to the year 2000.

Having identified the key component types, the next stage of the process is to identify how the different component groups interact to form the various configuration abstractions. Once these configurations have been established the process can be further extended by defining a mechanism for describing the properties or attributes of each of the components groups and their relationships.

4.4.3 ISCM Relationships

In addition to identifying the component types of an application it is vital to ascertain the relationships existing between each of the component types and within particular component types, if unwanted actions such as the ripple effect are to be reduced. ISCM involves modelling a given application system at a number of different levels. Through doing this it is possible to define a series of software system configurations at different abstractions which can be used to assist the program comprehension process.

It is necessary to model a system at two fundamental levels within which the sub-levels of modelling are encompassed. These levels are:

- Intra-application modelling
- Inter-application modelling

4.4.3.1 Intra-application modelling (R_{intra})

This level of modelling is concerned with the identification of relationships existing between application components of the core configuration and as such may be represented as:

$$\textit{application} \quad \Leftrightarrow \quad \textit{application} \quad (RCg_{app\text{-}app})$$

This is the traditional view of a system configuration and is the level at which most of the program comprehension to date has been concentrated. Additionally, within this level the modelling may be further divided into:

- Intra-component modelling.
- Inter-component modelling.

Intra-component modelling is now a common approach to program comprehension. In this mode, modelling occurs at a very fine level of granularity, concentrating on the dependencies and linkages occurring within a single module or file, that is, comprehension at the functional, procedural or even variable definition level. Techniques such as call graph derivation, program slicing and variable cross referencing are all common strategies for dealing with intra-component or low level modelling and program comprehension.

Inter-component modelling concentrates on the dependencies and linkages occurring between application components, that is, at the module or file level of granularity. In the context of this thesis, it is these inter-component dependencies that are of most interest. For this reason, little emphasis will be placed on modelling below the entire component level and hence no distinction will be made between the intra-application modelling types.

4.4.3.2 Inter-application modelling (R_{inter})

This level of modelling is concerned with the identification of relationships existing between components of the core application configuration and components of the other types identified in Section 4.1. The following relationship groupings have thus been identified:

application	$\Leftrightarrow$	*versioned*	$(RCg_{app\text{-}ver})$
application	$\Leftrightarrow$	*domain*	$(RCg_{app\text{-}dom})$
application	$\Leftrightarrow$	*incremental*	$(RCg_{app\text{-}cog})$
application	$\Leftrightarrow$	*documentary*	$(RCg_{app\text{-}doc})$
application	$\Leftrightarrow$	*user*	$(RCg_{app\text{-}user})$
application	$\Leftrightarrow$	*interface*	$(RCg_{app\text{-}int})$
application	$\Leftrightarrow$	*data*	$(RCg_{app\text{-}data})$
application	$\Leftrightarrow$	*associated*	$(RCg_{app\text{-}assoc})$
application	$\Leftrightarrow$	*allied*	$(RCg_{app\text{-}alli})$
application	$\Leftrightarrow$	*environmental*	$(RCg_{app\text{-}env})$
application	$\Leftrightarrow$	*third_party*	$(RCg_{app\text{-}trd})$
application	$\Leftrightarrow$	*enabling*	$(RCg_{app\text{-}enab})$
application	$\Leftrightarrow$	*alien*	$(RCg_{app\text{-}alien})$
application	$\Leftrightarrow$	*missing*	$(RCg_{app\text{-}miss})$

There may also be relationships existing between components groups other than with those of the application group. However, the extra level of complexity arising from these interactions is considered to be outside the scope of this thesis. It is considered to be a topic for further investigation and hence will not be modelled here.

The identification of the component types and the relationships existing between the application and other component groups leads to the definition of a number of configuration abstractions for any system under investigation. These are represented generically in Figure C4-11 and more specifically in Figures C4-12 to C4-19. The membership of each configuration example is also described using a simple set theory.

4.4.4 ISCM Configuration Abstractions

Abstraction is a technique for handling complexity in real-world scenarios by masking irrelevant details for defined situations [111]. It thus follows that understanding of the complexities of software systems during the program comprehension process could be aided through the use of abstraction to selectively model different aspects of a system. Indeed, abstraction is not a new phenomenon with regard to the construction of new or reverse engineering of old systems. However, prior approaches have tended to concentrate on abstracting contexts and actions for source level understanding or process concepts in order to recover design decisions. There is thus scope for investigating how 'layers' of a system can be abstracted or 'peeled away' in order to recover the components and organisation of a software configuration. Indeed, by using carefully selected patterns of requested information, the component groups and relationships existing between them can be systematically modelled until representation of the entire system has been achieved.

The previous sections have described the component types and the relationships existing between the core application system and the other component groups. This section will present a number of possible configuration abstractions. As discussed in Chapter 3, the primary abstractions supported by the ISCM process are at the baseline, program family and environmental levels. However, any number of abstractions may be identified according to the needs of a system under investigation. Indeed, the exact nature of component groupings and their combination into configuration abstractions has a degree of subjectivity associated with it. The ISCM process intentionally supports this flexibility in order to exploit one of the most powerful concepts of abstraction, that is, being able to make visible an exact area of interest within a system at the level of detail required. The generic framework for building up a configuration is shown in Figure C4-11 and examples of specific configuration abstraction patterns in Figures C4-12 to C4-19.

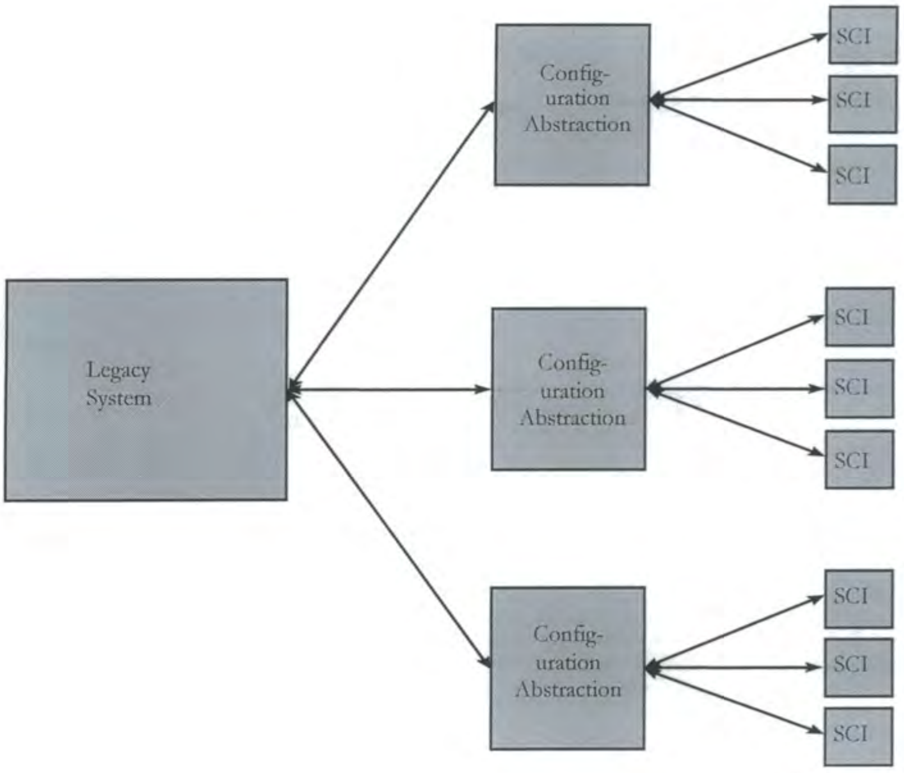

Config-uration Abstraction
SCI
SCI
SCI
Legacy System
Config-uration Abstraction
SCI
SCI
SCI
Config-uration Abstraction
SCI
SCI
SCI

Figure C4-11 Generic configuration build-up

The membership of each configuration abstraction may be described using a purposely simplified set theory where:

$\bigcup$	is the union of all single elements (either components or relationships) that are associated with a particular application at the specified configuration abstraction.
$\cup$	is the union of sets (component types or relationship types) that are applicable to the specified configuration abstraction.
C	is a component within the defined (component type) set.
R	is a relationship between two components of the defined set or sets.
$\leftrightarrow$	is a bi-directional linkage to another application component
$\mapsto$	is a uni-directional linkage to a component of a different type.
+	is additive, representing the sum of two different sets (i.e. a set of components and a set of relationships).

4.4.4.1 Total

The *total* configuration abstraction models an entire application. As such it may contain components in any or all of the identified component groups. Figure C4-12 demonstrates the necessity for a series of abstractions if a software system is to be clearly modelled. It is extremely difficult to identify the different component parts of the configuration when viewed as a whole. However, this is the situation commonly facing the personnel charged with maintaining legacy code. Maintainers need to be able to progressively identify the different components and to map the relationships of the application components to each other and to the components in other groups.

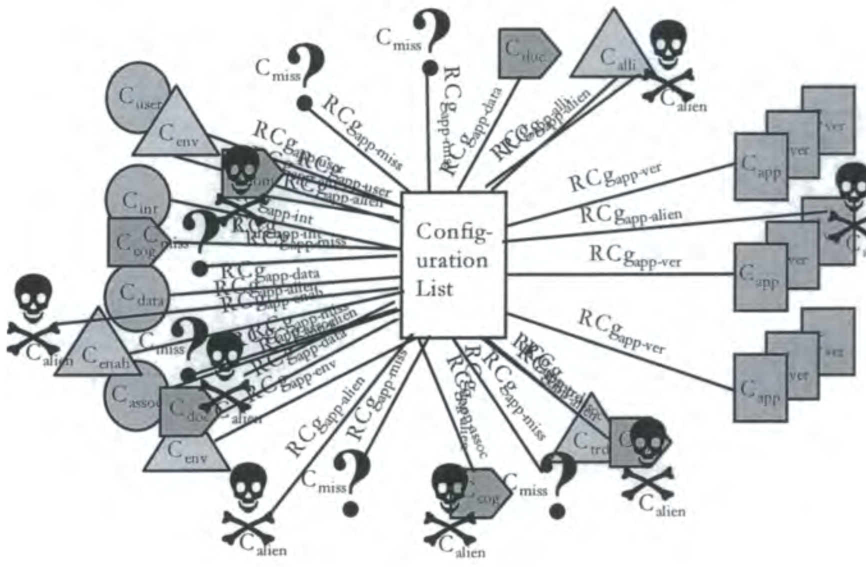


Figure C4-12 Total configuration abstraction

The total configuration may be represented as:

$$\mathbf{C}_{\text{total}} = \bigcup (\{C_{app}\} \cup \{C_{ver}\} \cup \{C_{dom}\} \cup \{C_{cog}\} \cup \{C_{doc}\} \cup \{C_{user}\} \cup \{C_{int}\} \cup \{C_{data}\} \cup \{C_{assoc}\} \cup \{C_{alli}\} \cup \{C_{env}\} \cup \{C_{trd}\} \cup \{C_{enab}\} \cup \{C_{alien}\} \cup \{C_{miss}\}) + \bigcup (\{RC_{app\leftrightarrow app}\} \cup \{RC_{app\mapsto ver}\} \cup \{RC_{app\mapsto dom}\} \cup \{RC_{app\mapsto cog}\} \cup \{RC_{app\mapsto doc}\} \cup \{Rc_{app\mapsto user}\} \cup \{Rc_{app\mapsto int}\} \cup \{RC_{app\mapsto data}\} \cup \{RC_{app\mapsto assoc}\} \cup \{RC_{app\mapsto alli}\} \cup \{RC_{app\mapsto env}\} \cup \{RC_{app\mapsto trd}\} \cup \{RC_{app\mapsto enab}\} \cup \{RC_{app\mapsto alien}\} \cup \{RC_{app\mapsto miss}\})$$

4.4.4.2 Baseline

The *baseline* configuration abstraction models the traditional view of a software system configuration. This is essentially the core application components and how they relate to each other. The composition of the baseline configuration is shown in Figure C4-13.

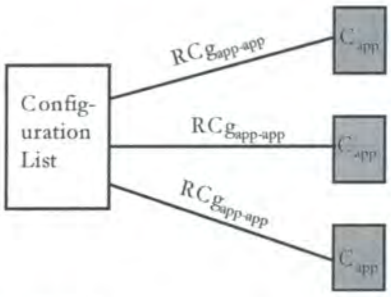


Figure C4-13 Baseline configuration abstraction

The baseline configuration may be represented as:

$$\mathbf{C}_{\mathbf{baseline}} = \bigcup (\{C_{app}\}) + \bigcup (\{RC_{app \leftrightarrow app}\})$$

4.4.4.3 Program Family

The *program family* configuration abstractions can be used to model versioning at a number of levels. For example, Figure C4-14 shows the set of version groups existing for the baseline configuration. Alternatively, abstractions can be used to model versioning within the primary or secondary environments if this level of detail is required, or they can be used to model a particular (single-version) application release within a program family. Alternatively, within a controlled system, the set of (multi-version) baseline abstractions can be modelled over time to record the developmental or maintenance history of a system at an entire configuration, rather than solely individual component, level.

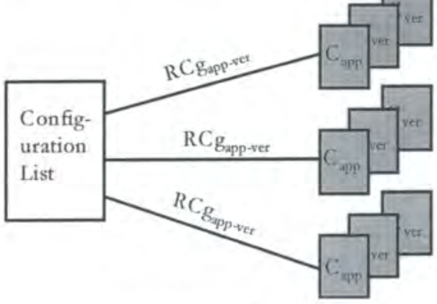


Figure C4-14 Versioned (baseline) configuration abstraction

The baseline family configuration may be represented as:

$$\mathbf{C}_{\mathbf{versioned(baseline)}} = \bigcup (\{C_{app}\} \cup C_{ver}) + \bigcup (\{RC_{app \leftrightarrow app}\} \cup \{RC_{app \to ver}\})$$

4.4.4.4 Environmental

The *environmental* abstraction models the interactions occurring between the application components and those of the surrounding system environment. A distinction has been made between the primary and secondary environment based on the nature of the component relationships. Essentially, the primary environment is composed of those components with which the application components have a direct relationship in terms of their required integration or interaction to enable execution of the software. The primary environment therefore models component types which include the user, interface facilities, data and associated system and application libraries. The primary environment is modelled in Figure C4-15.

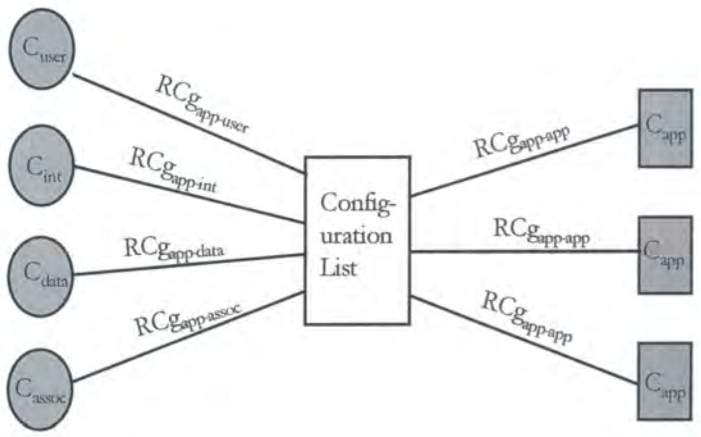


Figure C4-15 Primary environment configuration abstraction

The primary environmental configuration may be represented as:

$$\mathbf{C}_{\mathbf{primary}} = \bigcup\ (\{C_{app}\} \cup \{C_{user}\} \cup \{C_{int}\} \cup \{C_{data}\} \cup \{C_{assoc}\}) + \bigcup\ (\{RC_{app\leftrightarrow app}\} \cup \{Rc_{app\mapsto user}\} \cup \{Rc_{app\mapsto int}\} \cup \{RC_{app\mapsto data}\} \cup \{RC_{app\mapsto assoc}\})$$

The secondary environment is composed of those components which enable construction or change of the baseline application but which are not actually maintained or incorported as part of it. Thus components such as tools for compiling and linking the application, tools for maintenance information extraction, third party tools, operating system facilities and hardware components can be modelled within this abstraction. The secondary environment is shown in Figure C4-16.

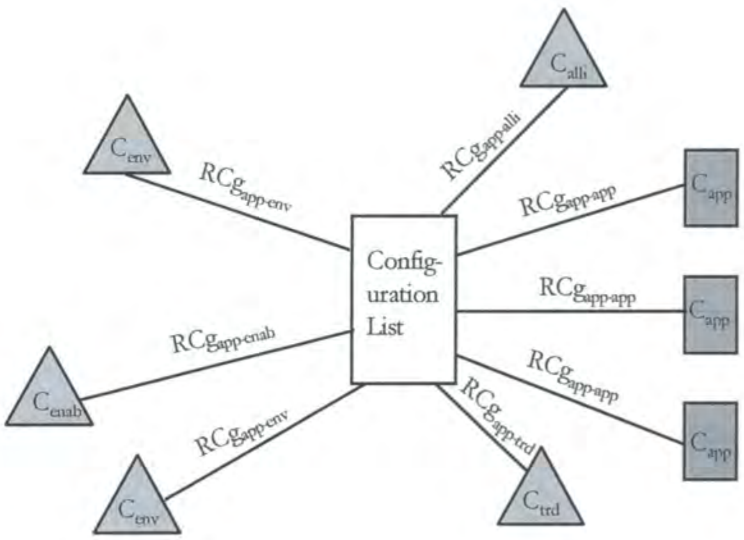


Figure C4-16 Secondary environment configuration abstraction

The secondary environmental configuration may be represented as:

$$\mathbf{C}_{secondary} = \bigcup \left(\{C_{app}\} \cup \{C_{alli}\} \cup \{C_{env}\} \cup \{C_{trd}\} \cup \{C_{enab}\}\right) + \bigcup \left(\{RC_{app\leftrightarrow app}\} \cup \{RC_{app\mapsto alli}\} \cup \{RC_{app\mapsto env}\} \cup \{RC_{app\mapsto trd}\} \cup \{RC_{app\mapsto enab}\}\right)$$

4.4.4.5 Documented

The *documented* abstraction models the relationships existing between the baseline application and any documentation created during the development or maintenance processes. As such the documents may: reflect the products of the development stages such as requirements or design documents, test plans, test data etc.; include documented quality assurance activities such as review proceedings; or record knowledge reclaimed about an application during the maintenance process. The composition of the documentation configuration is shown in Figure C4-17.

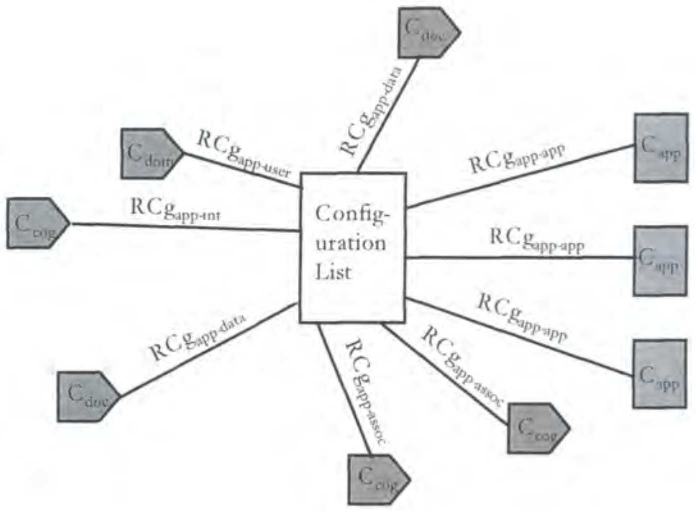


Figure C4-17 Documented configuration abstraction

The documented configuration may be represented as:

$$\mathbf{C}_{\mathbf{documented}} = \bigcup\ (\{C_{app}\} \cup \{C_{dom}\} \cup \{C_{cog}\} \cup \{C_{doc}\}) + \bigcup\ (\{RC_{app\leftrightarrow app}\} \cup \{RC_{app\mapsto dom}\} \cup \{RC_{app\mapsto cog}\} \cup \{RC_{app\mapsto doc}\})$$

4.4.4.6 Anomalous

The *anomalous* abstractions relate to the modelling of configurations which have become corrupted or masked through a dominance of unrelated or alien components, or which have become incomplete through the loss or misplacement of essential components. The composition of the corrupted configuration is shown in Figure C4-18. These components need to be flagged to the maintainer as candidates for archival or removal from the application framework. This is because, even if these components do not directly affect the correct operation of the system, their presence can greatly hinder the program comprehension process.

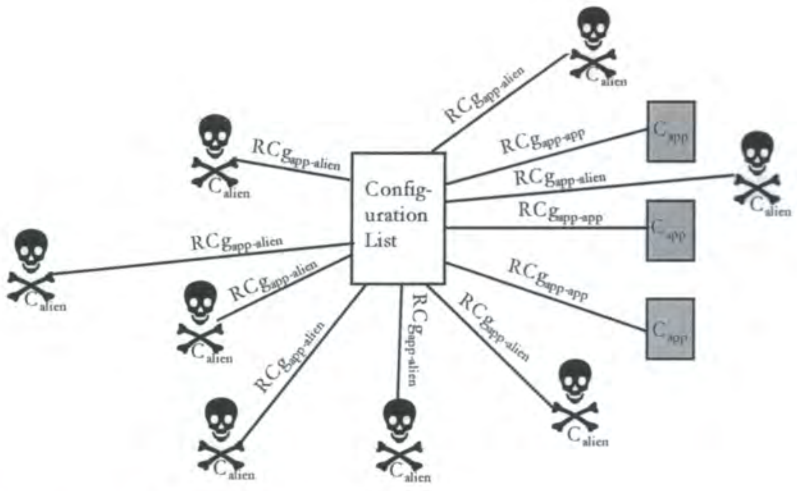


Figure C4-18 Corrupted configuration abstraction

The corrupted configuration may be represented as:

$$\mathbf{C}_{\mathbf{corrupted}} = \bigcup(\{C_{app}\} \cup \{C_{alien}\}) + \bigcup(\{RC_{app\leftrightarrow app}\} \cup \{RC_{app\mapsto alien}\})$$

Conversely, any components that have been flagged during the ISCM process as missing or misplaced from the application directories must be 'found' and replaced within the application framework. If after searching they cannot be located, they must be re-created in order to maintain the correct operation of the application across the entire range of its functionality. The incomplete configuration abstraction is shown in figure C4-19.

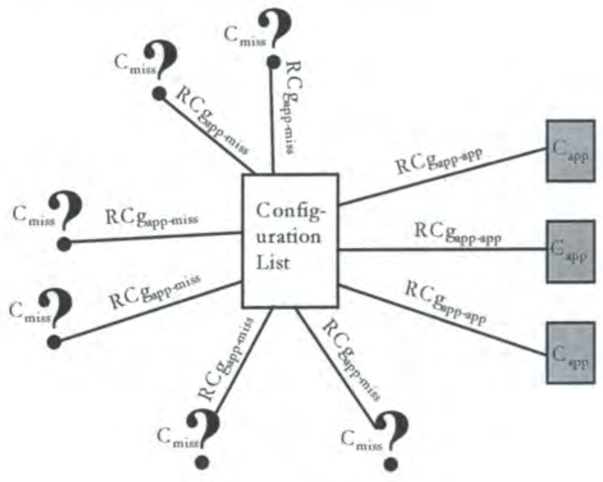


Figure C4-19 Incomplete configuration abstraction

The incomplete configuration may be represented as:

$$\mathbf{C}_{\mathbf{incomplete}} = \bigcup(\{C_{app}\} \cup \{C_{miss}\}) + \bigcup(\{RC_{app\leftrightarrow app}\} \cup RC_{app\mapsto miss}\})$$

4.4.5 Attributes & Storage of Components

As mentioned in Section 4.2, the ISCM process places an increased emphasis on the identification phase of the SCM process. It is therefore imperative to be able to document either immediately or incrementally, a full complement of information regarding each particular component and how it is integrated into one or more configurations. This can be achieved through the identification of a set of component attributes which in combination will uniquely and fully identify a component and any corresponding configuration compositions. From studies of the literature and a number of 'real-world' systems such an attribute set has been defined for use within the ISCM process. Some of the data related to these attributes is immediately evident from the system and some can be generated automatically through use of the host resident tools, but the remainder will need to be recovered and recorded during the maintenance process itself. In this way the level of component and configuration identification can parallel or indeed exceed that recorded during the traditional development-oriented SCM process. The attribute set of a component upon which the ISCM identification activity is based is shown in figure C4-20 and corresponds to the **/{Components}/** level of the Inverse Configuration Description Language (ICDL) shown in Table 4-3.

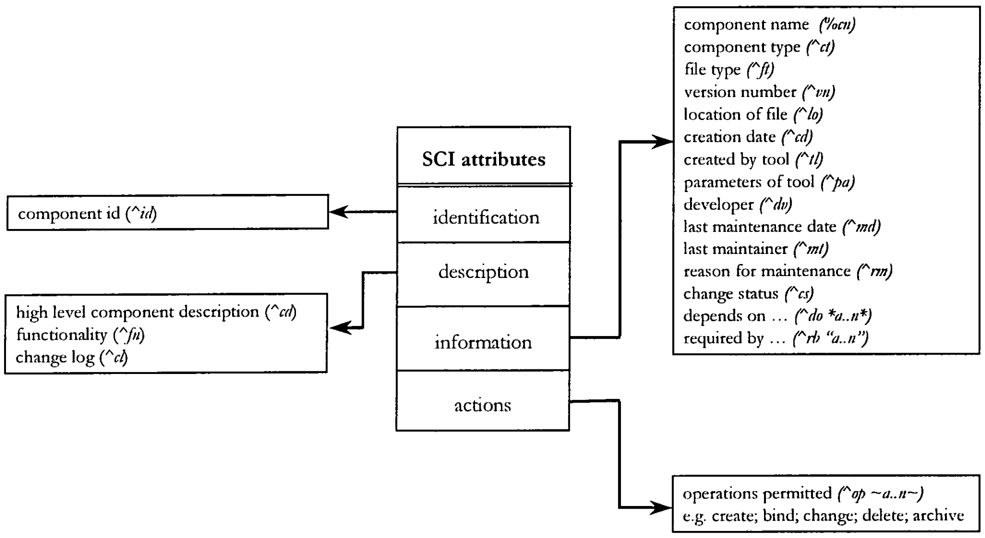


Figure C4-20 Component attribute set

All configuration and component information, together with rules defining ways to collect and collate this information from different application and system types must be stored within the Extensible System Information Base. A number of approaches and possible structures for the storage of this information were discussed in Chapter 3, and the actual structures chosen are described in Chapter 5 within the context of the PISCES M^4 system.

The foundation of the ISCM process and its associated method is thus the representation of software systems architectures by means of an Extensible Systems Information Base (ESIB). Within the scope of this work the characteristics of a number of Unix, MS-DOS and Windows95 systems and their applications were studied (see Chapter 7) and architectural models manually built-up for each application. The characteristics of the separate models were then used in combination to create the generic model and to identify a set of initial rules for information reclamation. This information was subsequently used for seeding the knowledge base. However, it is intended that the ESIB will be incrementally extended as additional information about specific domains, system types and applications is obtained. Through this principle it should be possible to approach the premise of complete genericity of the model.

The ESIB thus contains information regarding key features of operational environments such as operating systems, systems architectures, application types, tools and programming. This information can be held as records or 'objects' to represent the generic view of a system. This generic view can then be instantiated as a tailored and subsequently specific view for a particular system as more information is gained about the system. The information collection process is facilitated through the definition of a number of proformas or templates which can be progressively populated with information of a general or more specific nature. The structure and detailed role of these proformas is discussed in Section 4.6.

4.5 The Inverse Configuration Description Language (ICDL)

This section describes the development of the ICDL, the configuration modelling language for the ISCM process. As mentioned in Chapter 3, most configuration languages are targeted at the PITL level and as such they are primarily concerned with the construction of systems from their components parts, either during development or after changes have been made during maintenance. Whilst the construction of a correct build remains as the ultimate goal, the ISCM process is primarily addressing the program comprehension activity of the maintenance process which occurs prior to any changes being made. Correspondingly, within the context of the ISCM process, the principal emphasis is on extending the modelling of software system architectures or configurations to encompass environment, domain and cognitive components. These extended models will enable the system to be more readily understood prior to the change being made and the new software builds occurring. The ICDL is therefore considered to be a programming-in-the-environment (PITE) language, which it achieves by building upon and extending a combination of features taken from several of the PITL languages described in Chapter 3.

4.5.1 Structure of the ICDL

When devising the ICDL for the purpose of PITE, two possible approaches were considered. Firstly, the ICDL could be developed as an all encompassing system-build language encompassing PITL and PITS constructs as well as facilities for describing systems at a PITE level. Alternatively, the ICDL could be developed exclusively as a PITE language which could then act in combination with PITL and PITS languages as required. The latter of the two approaches was considered to be the more appropriate for reasons of flexibility, scope of work and relevance to the issue being solved, primarily that of recovering and documenting the configurations of viable builds (the inverse configuration management approach) rather than enabling and verifying builds from a series of separate component parts (the traditional configuration management approach) .

Structurally the ICDL consists of seven levels or abstractions of system information. An overall system identifier and date field are also included to enable a historical record of the evolution of the system to be constructed. These levels may be defined as:

- ***Domain***: this level documents the features and characteristics of the general environment and domain (*domain knowledge and experience*).

- ***Environmental***: this level defines the general computing characteristics of the application system (*operational specification and characteristics*).

- ***Location***: this level defines the location of the key libraries, component type directories and tools pertaining to a system type or application (*physical specification and characteristics*).

- ***Total:*** this level documents all of the components that are or that may be associated with a particular system architecture. This includes any present but apparently redundant or 'alien' components and a record of any components which are referenced as being associated with a software architecture but which appear to be missing. Through this approach, the total level documents unselectively the application and its associated environment (*unfiltered information*).

- ***Abstraction***: this level defines the components that are considered to be part of each configuration abstraction and as such acts as the driver for determining 'what' and 'how much' component information should be extracted from the knowledge base pertaining to a particular system (*definition of parameters for filtering of information*).

- ***Configuration***: this level documents the combination of components associated with a software system architecture or configuration as defined by the level of abstraction selected (*filtered information at defined level of abstraction*).

- ***Component***: this level documents the attributes and relationships (see figure C4-20) of individual components, that is, it defines the resources provided and required by a module and an overall description of the implementation part of the components. It does not however, deal in detail with implementation details, unless these have been recovered and incrementally documented as part of the reclaimed understanding about a system (*more detailed component identification and comprehension information*).

Syntactically the information contained within each level may be represented as a sequence of simple commands. Parsing of this information by bespoke or proprietary utilities enables the relevant sections of information about a system to be extracted and passed to other tools for manipulation or reporting purposes. The syntax of the language and a description of each of the relevant identifers is given in Table C4-2. A discussion of the usage of the language for documenting software systems is given in Chapter 7.

As can be seen from Table C4-2, the lexical and syntactical approach of the ICDL is targeted towards machine- rather than human-processing in terms of its simplicity and lack of the elaborate record structures evident in some description languages. Never-the-less the ICDL encompasses all the essential features necessary to describe system architectures and it does so in such a way that its representation is compatible across multiple platforms and can be readily updated and extended. Additionally, with a little practice and instruction maintainers should very easily be able to directly decipher the information contained in the resultant file. In these respects the ICDL approach adopted is similar to that of the DOCMAN intermediate file format [58], although the record structures are considerably different in their structure and content. The simplicity of the ICDL language also enables easier integration with the human-readable PICS although this is currently at a conceptual and manual level rather than by automatic conversion.

The ICDL is able to incrementally document a wide variety of information pertaining to the reconstruction of software system configurations such that it encompasses cognitive, domain and environmental information in addition to the more traditional configuration management identification and dependency information. The levels of information extracted or presented about a particular system may be controlled via the abstraction section of the ICDL. This section also enables new abstractions to be defined in relation to a particular system or application domain.

/Header/	
%sn:	Overall ***system name***
%dt:	***Date*** that configuration information was generated.
%pi:	Proforma identifier (linked to specific proforma)
%pv:	Versioned proforma identifier (linked to specific proforma)
/Domain/	
%da:	Generic ***domain application*** area of the system.
%st:	***System type***
%gd:	***General description*** of what the system is intended to do.
%si:	Any ***specific issues*** that need to be flagged such as safety issues, standards to conform to etc.
/Environment/	
%pl:	Application host system ***platform***
%ar:	Overall application ***architecture***
%os:	Application ***operating system***
/Location/	
%sl:	***System libraries***
%vc:	***Version control*** system
%df:	***Data files***
%et:	***Extraction tools***
/Total_configuration/	
%mc:	***Master configuration*** list
^{component/cn}	Iteration of ***component name(s)***
/Abstraction/	
%bs:	***Baseline*** abstraction
%pf:	***Program family*** abstraction
%en:	***Environmental*** abstraction
%dc:	***Documented*** abstraction
%an:	***Anomalous*** abstraction
/{Abstracted_configuration}/	
%ct:	Abstracted ***configuration title***
%cl:	Abstracted ***configuration list***
^{component/cn}	Iteration of ***component name(s)***
/{Component}/	
%cn:	***Component name***
^id:	***Component id***
^ct:	***Component type***
^ft:	***File type***
^cd:	High level ***component description***
^fn:	***Functionality***
^vn:	***Version number***
^lo:	***Location*** of file
^cd:	***Creation date***
^tl:	Created by ***tool***
^pa:	***Parameters*** of tool
^dv:	***Developer***
^md:	Last ***maintenance date***
^mt:	***Maintainer***
^rm:	***Reason*** for ***maintenance***
^cl:	***Change log***
^cs:	***Change status***
^do:	***Depends on***
a..n:	Component(s) ***a .. n***
^rb:	***Required by***
"a..n":	Component(s) ***a .. n***
^op:	***Operations*** permitted
~a..n~:	Operation(s) ***a .. n***

Key:

/ /	Section level marker	{ }	Iterated component or information
%	Outermost-level data	^	Sub-level of data
* *, " ", ~ ~	Sub-sub-list of data		

Table 4-3 ICDL syntax and command structure for PITE

Storage of information extracted using the ICDL is currently held in a flat-file arrangement, although parts of the M^4 prototype system use a relational approach in order to enable more flexible search and retrieval patterns to be defined and more efficient storage of 'chunks of information' or structured object types to be carried out.

4.5.2 System Description Document

The basis of a configuration is typically the system description document (SSD) which acts as a blueprint for the construction of a system. Examples include *Make's* [146] *makefile*, *Shape's* [280] *shapefile*, *Adele's* [135] *system model*, *Odin's* [90] *compiled knowledge base* and *Gypsey's* [95] *configuration template*. *Jasmine* [258] also uses a *template* to define the system model. The system model essentially contains all the 'knowledge' about a system. This may include, all software objects or references to objects, relationships between objects, transformation rules, selection rules, tool parameters and variant definitions. In these respects the system description documents are often a more detailed representation of the systems they describe than those of the configuration description languages which tend to focus on the PITL concerns of module interface definition, programming resource requirements and the subsequent combination of the modules into an executable system.

Chapter 3 has outlined a number of languages for describing syntactically software system architectures/configurations and this chapter has extended these principles, to more fully describe the system configurations and the complexity of their interactions with the environment, through development of the ***Inverse Configuration Description Language*** (ICDL). The ICDL thus provides the framework for representing information about system architectures and enables manipulation of these data by a number of data extraction and representation tools. However, from a human computer interaction (HCI) perspective the ***Proforma Increasing Complexity Series*** (PICS) provides a more comprehensive and intuitive interface for reclaiming and recording information about a software system. The role and format of the PICS is described in section 4.6.

4.6 Proforma Increasing Complexity Series

Information about each of the component client groups identified in section 4.4.2 is collected and stored in a system knowledge base which is then represented for a particular application as a ***Proforma Increasing Complexity Series (PICS)***. The series may be described by a number of proformas each one becoming more detailed and specific in terms of the application under study. Thus, each PICS represents a software system architecture at three levels of abstraction: generic, tailored and specific. As shown in Figure C4-21, the abstraction enables the configuration description document to move from being independent of any application and any application host environment towards being a totally application and host specific description or 'genetic fingerprint' of a software system. In this way the transition is made from an architectural model of a system to a configuration description or instantiation of a specific application and represented by the SSD. Additionally, if a series of genetic fingerprints of a particular system are taken over a period of time, the evolution of a system version or even an entire program family may be progressively built-up. The syntax of the ICDL that integrates with the PICS information can also be used as the intermediate file format for hypermedia manipulation of information pertaining to the software system.

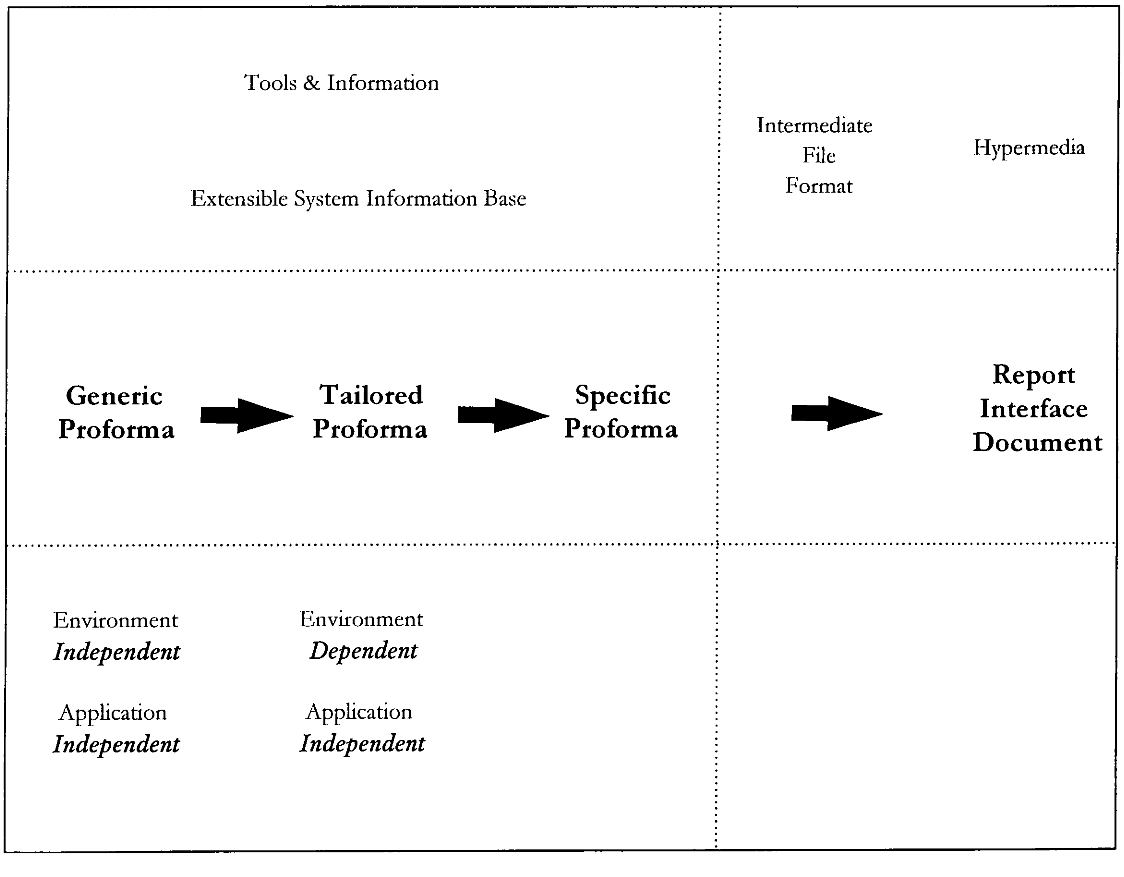
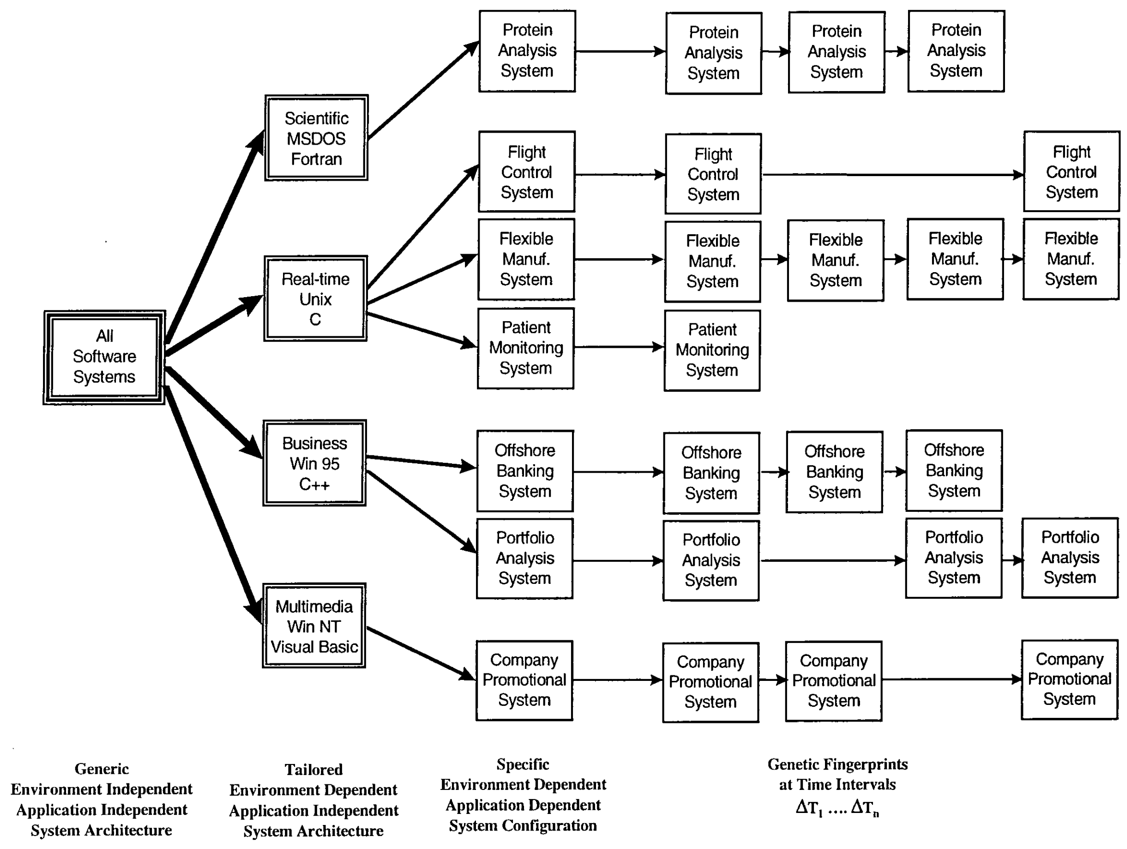


Figure C4-21 Independent vs. dependent implementation of the PICS

The aim of the PICS series within the ISCM process is to drive and ultimately semi-automate the program comprehension process. Initially however, or at least until a substantial quantity of data have been assembled within the *Extensible System Information Base* (ESIB), the program comprehension process must be conducted manually through system observation and incremental recording of reclaimed information. The comprehension process is however greatly aided and driven by the provision of the PICS which are essentially a defined set of proformas or templates that become cumulatively populated with system information as the comprehension process occurs each time a maintenance change is required.

As indicated in Figure C4-21, the PICS consists of three key levels of proforma abstraction which represent the transition from a *generic application independent system model* to an *instantiated application dependent configuration.* Due to the abstracted nature of the proformas there is not a 1 : 1 : 1 mapping of generic : tailored : specific proformas. The actual mapping is represented pictorially in Figure C4-22 and is explained in the remainder of this section together with a description of the role of each of the proformas.


All Software Systems
Scientific MSDOS Fortran
Real-time Unix C
Business Win 95 C++
Multimedia Win NT Visual Basic
Protein Analysis System
Flight Control System
Flexible Manuf. System
Patient Monitoring System
Offshore Banking System
Portfolio Analysis System
Company Promotional System
Generic Environment Independent Application Independent System Architecture
Tailored Environment Dependent Application Independent System Architecture
Specific Environment Dependent Application Dependent System Configuration
Genetic Fingerprints at Time Intervals $\Delta T_1 .... \Delta T_n$

➨ represents refinement of generic proforma into tailored proforma
➔ represents refinement of tailored proforma into specific proforma
→ represents refinement of specific proforma over time as maintenance proceeds

Figure C4-22 Relationship between generic, tailored and specific proformas

4.6.1 *Abstraction_Level_1* : Generic proforma

The generic proforma is *environment independent* and *application independent.* Its role is to define the information required by all systems, whatever the application domain, target platform, operating system or language, and to specify the information required for selection/creation of the tailored proforma. Due to the generic nature of the proforma there is likely to be only one instance of the generic proforma in connection with a particular ESIB.

4.6.2 *Abstraction_Level_2* : Tailored proforma

The tailored proforma is *environment dependent* but *application independent.* Its role is to define what information needs to be obtained in order for a category of applications to be modelled within the defined criteria of a particular tailored proforma. It also specifies the information that needs to be extracted in order to generate the specific proforma and defines more specifically the tools that should be available on the host system to assist in the generation of application specific information. As a tailored proforma can collectively model a number of systems at this level of abstraction it is likely that several instances of the tailored proforma will exist in connection with a particular ESIKB. As more systems are maintained and more application scenarios combinations are recorded, the number of tailored proformas will increase. For systems matching the attributes of a tailored proforma, a 'later' starting point into the program comprehension process may be enabled.

4.6.3 *Abstraction_Level_3* : Specific proforma

The specific proforma is *environment dependent* and *application dependent.* Its role is to describe and define unequivocally the application under investigation in terms of the component parts that make up the system configuration and the relationships existing between these components and their environmental counterparts. Any knowledge regained and documented about the system during the program comprehension process should also be evident at this stage. It is at this level that information regarding the application components, associated versions, missing components, redundant components, allied and environmental components is revealed. As the specific proforma is unique (fingerprinted) to a particular application system, there will be at least one proforma per application and possibly several if the proformas are baselined over a period of time. Baselined proformas enable the evolutionary history of a system to be determined either by consultation of the baselines themselves, or via a report consisting of the cumulative deltas which provide a time-ordered series of change-events made to the system. Information contained at this level of proforma abstraction corresponds closely to the ICDL representation of the software configuration.

4.6.4 Proforma Information

Each of the proformas is divided into six major sections or levels with regard to the type of information to be recovered and documented. These sections have been designed to map onto the ICDL constructs defined in section 4.5 such that the information in the proformas is also reflected within the ICDL. The exact fields included may be tailored to be representative of the type of system being maintained.

4.6.4.1 *Section_Level_1* : Header (identification section)

This section provides the identifying information for the system being maintained. The key fields in this section are:

- *The name of the system being maintained*
- *The date the proforma was raised or updated*
- *A unique identifier based on [proforma abstraction level + incremental counter]*
- *A versioned identifier based on [unique identifier + revision number]*

The purpose of maintaining this information is twofold, firstly it enables the system being documented using the PICS to be easily identified and secondly at the specific proforma level it enables a maintenance/evolutionary history of the system to be constructed.

4.6.4.2 *Section_Level_2* : Environment and domain (descriptive section)

This section contains information regarding the application domain characteristics and the general features of the application operating environment. The key fields encompassed are:

- *Application domain area (Manufacturing, Finance, Commercial, Robotics etc.)*
- *Application system type (Real-time, Batch, Event-driven, Embedded, etc.*
- *General description of the application system (Short paragraph describing key features of the system)*
- *Specific issues for concern (Performance, Safety, High volume throughput etc.)*
- *Platform (PC, Workstation, Mainframe etc.)*
- *Architecture (Distributed, Standalone, Centralised etc.)*
- *Operating System (Unix, MSDOS, VMS, Windows95, WindowsNT etc.)*
- *Language (C, Pascal, C++, Delphi, COBOL, Java, J++ etc.)*

Maintenance and specification of these items of data enable the proforma to be tailored towards a particular type of system. Additionally, it is the information contained within this section that determines whether a new tailored proforma needs to be raised for a particular system undergoing maintenance.

4.6.4.3 *Section_Level_3*: Resource and location (definition section)

This section contains information regarding the resources, tools and utilities available on the host system platform. It also defines the default or specified location of the different component types associated with an application. The following types of information are specified:

- *Location of system libraries*
- *Location of system tools used*
- *Location of tools available for data extraction*
- *Version control system used (.bak, rcs, sccs etc.)*
- *Location of application tools and libraries*
- *Location of application source files*
- *Location of data files*
- *Location of documentation*
- *Location of test files*
- *etc.*

4.6.4.4 *Section_Level_4* : Configuration abstractions (specification section)

This section contains the definitions of the component types or groups that constitute each of the configuration abstractions defined in section 4.4.4. The abstractions act as the driver for determining 'what' and 'how much' information should be studies and recorded with respect to a particular system. The key abstractions supported are:

- *Total (application + versioned + domain + cognitive + user + documentary + interface + data + associated + allied + alien + environmental + third-party + enabling + missing components).*
- *Baseline (application components)*
- *Program family (application + versioned components)*
- *Environmental (application + versioned + domain + user + interface + data + associated components)*
- *Documented (application + domain + cognitive + documentary components)*
- *Anomalous (application + alien + missing components)*

Whilst the above categories have been defined as the default configuration abstractions, the components identified as making up each abstraction may be redefined or new abstractions added within the proforma in response to the needs of the system being maintained. This allows a high degree of tailoring and flexibility in the way the configurations are defined. Additionally, by implication the abstraction definitions can also be interpreted as the intra- and inter-relationships that will need to be identified as existing between the application components themselves and the application components and other component groups.

4.6.4.5 *Section_Level_5*: Component identification (extraction section)

This section contains information regarding the components present that make-up the application and its environment such as source files, header files, synthesised files, executable files, object files, archived files, libraries; archived libraries, documentation files, documentation files, test files, data files, tools used, versions, presumed missing files, presumed redundant files; presumed 'alien' files. Each file type is recorded against the most appropriate component group as identified in section 4.4.2, however this categorising of files into component groups may again be redefined within the proformas in response to the specific needs of the system being maintained. The configuration abstractions and component identification thus combine to form a filtering mechanism to enable a more focused study on the relevant views of the application being maintained. The key component groups defined are:

- *Application (source + header + object + executable + synthesis components etc.)*
- *Versioned (archived files + archived libraries + rcs + sccs + back-up components etc.)*
- *Domain (documentation + annotation + audio + video + graphic + animation components etc.)*
- *Cognitive (documentation + annotation + audio + video + graphic + animation components etc.)*
- *Documentary (documentation + text + rich-text + hypertext components etc.)*
- *User (programmer + maintainer + manager + customer components etc.)*
- *Interface (widgets + foundation classes + interface library + linkage components etc.)*
- *Data (database + spreadsheet + analysis information + flat-file components etc.)*
- *Associated (application library components + test suite components etc,)*
- *Allied (general system libraries + operating system components etc)*
- *Third-party (rapid application development tools + databases + performance analyser components etc.)*
- *Environmental (compiler + editor + linker components etc.)*
- *Enabling (version control tool + data extraction tool + drawing tool components etc.)*
- *Alien (any components without primary or secondary connections with the application)*
- *Missing (primarily application components but may be associated with an component group)*

The allocation of components types to their respective groups is to an extent subjective, thus within the proformas the components types within the groups may be extended or restricted as needs arise. Additionally as the proformas move from a generic to specific abstraction, the nature of the component definitions also change from being general file types to specifications of singular files.

4.6.4.6 *Section_Level_6* : Components (recorded/deduced)

This section contains the detailed information regarding each of the components identified at the varying levels of abstraction in Section_Level_5. The section contains both factual and recovered information about each component as well as details of their interactions with and dependencies on other application and environmental components. The key attributes that are recorded for each component are:

- *Component name (system specific name or identifier)*
- *Unique component identifier (shortened component name + type indicator + incremental counter)*
- *Component type (application, cognitive, domain, user, versioned etc.)*
- *File type (text, graphics, video, audio, rich-text, text etc.)*
- *High level component description (Brief description of component function)*
- *Version number (revision or variant number)*
- *Creation date (original creation date of component)*
- *Creator (original developer of component)*
- *Tool created by (enabling tool, compiler, linker etc.)*
- *Tool parameters (switches and options passed to the tool)*
- *Last maintenance date (date component last changed)*
- *Last maintainer (last person to change the system)*
- *Reason for last maintenance (reason for last maintenance change)*
- *Current maintainer (person assigned to proposed change)*
- *Reason for proposed maintenance (reason for current change)*
- *Depends on (components that call/ link to this component)*
- *Required by (components called by/ linked to this component)*
- *Operations permitted (edit, delete, create etc.)*
- *Location (where within the host system the component is physically located).*

The attributes defined in this section should be completed for each component that is studied as part of the program comprehension process, thereby conducting a thorough identification of each of the components contributing to the system configuration.

Tables C4-3 to C4-5 show the informational transition of the generic to the tailored to the specific proformas. Due to the large volumes of information produced even for a small application, for the purpose of demonstration, only a subset of the attributes in each of the sections of the proforma have been modelled. The change in emphasis towards information pertinent to a particular application should be evident from study of the completed proformas as they progress from *generic* to *tailored* to *specific* instantiations.

ABSTRACTED_LEVEL_1 : GENERIC

SECTION	*CRITERIA*	*CANDIDATES / RESULTS*
HEADER	System name	Any_software_system
	Proforma production date	22-05-1997
	Proforma identifier	G001
	Versioned proforma identifier	G001_v2
ENVIRONMENTAL	Domain area	manufacturing, medical, robotics, commerce, etc.
	System type	real-time, event-driven, embedded, etc.
	Operating system	Unix, MSDOS, VMS, Windows95, WindowsNT etc.
	:	
	Language	C, C++, Java, Pascal, FORTRAN, COBOL, Delphi etc.
DEFINITION & LOCATION	System libraries	/urs/lib/bin, C:\system\lib etc.
	Version control system	RCS, SCCS, .bak etc.
	Location of data files	~/rjmc/data, C:\MyDoc\Data etc.
	:	
	Extraction tools available	PCMS, Make, awk, grep, MMS etc.
ABSTRACTIONS	Baseline	application etc.
	Environmental	application, domain, versioned, user, interface, data etc.
	:	
	Anomalous	application, alien, missing etc.
IDENTIFICATION	Application components	source, object, header, makefiles, batch, autoexec etc.
	Domain components	documentation, annotation, graphics, animation etc.
	User components	programmer, maintainer, manager, customer etc.
	Enabling components	version control tool, data extraction tool, drawing tool etc.
	:	
	Alien components	any of the above
COMPONENT	Component name	Any_component
	Component identifier	any_req_001
	Component type	application, cognitive, domain, user, versioned, data etc.
	File type	text, graphics, audio, video, rich-text, text etc.
	High level component description	component function
	Depends on	object, executable, documentation, annotation
	:	
	Created by tool	Compiler, editor, linker

Table C4-4 The generic proforma

ABSTRACTED_LEVEL_2 : TAILORED		
SECTION	*CRITERIA*	*CANDIDATES / RESULTS*
HEADER	System name	Real_time_Unix_C
	Proforma production date	29-06-1997
	Proforma identifier	A003G001
	Versioned proforma identifier	A003G001_v2.1
ENVIRONMENTAL	Domain area	manufacturing
	System type	real-time
	Operating system	Unix
	:	
	Language	C
DEFINITION & LOCATION	System libraries	/urs/lib/bin.
	Version control system	SCCS
	Location of data files	~/rjmc/data/97
	:	
	Extraction tools available	Make, awk, grep.
ABSTRACTIONS	Baseline	application
	Environmental	application, domain, user, interface
	:	
	Anomalous	application, alien, missing
IDENTIFICATION	Application components	.c, .h, makefile, .obj, .exe
	Domain components	..txt, .doc
	User components	maintainer
	Enabling components	.exe
	:	
	Alien components	.pas, .doc, <22/7/95
COMPONENT	Component name	Component name
	Component identifier	AAA_NNN.req
	Component type	application, domain, user, interface
	File type	.txt., .c, .wav, .avi, MPEG, JPEG, .rtf
	High level component description	What the component does
	Depends on	.obj, .exe., txt, .ann
	:	
	Created by tool	C compiler

Table C4-5 The tailored proforma

ABSTRACTED_LEVEL_3 : SPECIFIC		
SECTION	*CRITERIA*	*CANDIDATES / RESULTS*
HEADER	System name	Flexible_manufacturing_system
	Proforma production date	07-07-1997
	Proforma identifier	S001A003G001
	Versioned proforma identifier	S001A003G001_v2.1.4
ENVIRONMENTAL	Domain area	manufacturing
	System type	real-time
	Operating system	Unix
	:	
	Language	C
DEFINITION & LOCATION	System libraries	/urs/lib/bin.
	Version control system	SCCS
	Location of data files	~/rjmc/data/97/fms/*.dat
	:	
	Extraction tools available	Make
ABSTRACTIONS	Baseline	application
	Environmental	application, domain, user, interface
	:	
	Anomalous	application, alien
IDENTIFICATION	Application components	control.c, control.h, makefile, control.obj,, …, control.exe
	Domain components	instruction.txt
	User components	rjmc, fjw
	Enabling components	awk..exe, Make
	:	
	Alien components	bank.pas, bank.doc
COMPONENT	Component name	control.c
	Component identifier	CON_001.src
	Component type	application
	File type	.c
	High level component description	Main controller for conveyer belt
	Depends on	fms.c, speed.c
	:	
	Created by tool	Borland C Compiler

Table C4-6 The specific proforma

4.7 ISCM Configuration Reclamation Process

In order to completely recover the system architecture/configuration of a legacy software system the ISCM process incorporates a number of activities similar to those described for the construction of software systems. For example, version groups to which components belong must be identified together with the correct revision or variant and any associated parameters or attributes that direct their combination into executing configurations. As much syntactical, semantic and cognitive information associated with a component must also be identified and made available if the partial or complete system undergoing maintenance is to be comprehended. However, there are also a number of fundamental differences between the forward and inverse processes, namely: for legacy systems the process tends to be much more iterative rather than sequential in nature. The end-product is also different in that the SCM construction of a system configuration involves moving from a system description to an executable system, while the ISCM identification of a configuration moves from an executing system to a description of the component parts and relationships existing between them. With regard to this, ISCM may initially be considered to act as a precursor to the software manufacture process. This can be attributed to the improved understanding gained through the ISCM activities of the system composition and its interaction with the environment. The resultant effect of ISCM is two-fold: firstly it enhances the probability of producing a viable build and secondly it reduces the chances of ripple effects propagating through the system.

It is additionally expected however, that as maintenance proceeds over the lifetime of a product the ISCM configuration description document can be built-up and controlled as part of the legacy system to such an extent that they can eventually be used themselves to drive the build process. At this point the forward and inverse approaches coincide and the system may be considered to be *'fully under control'* once more.

4.8 Summary

In order to address program comprehension issues of legacy systems, the ISCM process has been defined. Whilst ISCM incorporates the four key elements of the traditional SCM process, namely identification, control, status accounting and audit, it differs in the depth to which the identification phase of the process is treated within the ISCM process. The ISCM process model extends current work by enhancing existing maintenance models through the addition of detailed low-level process descriptions to guide information retrieval and understanding during the program comprehension activity. As such it is a prescriptive approach to the comprehension of a system, but one which maintains flexibility for customisation of individual working practices, together with identification of its configuration at varying levels of abstraction and degrees of formality is considered to be unique to the ISCM model.

To address the complexity of the software systems being developed and subsequently requiring maintenance, this chapter has also identified and defined a set of component types and groups from which an application and its environment may be modelled. Additionally, the relationships existing between the defined components may be modelled at two fundamental levels: intra-application modelling (between application components) and inter-application modelling (between the application and its environment). If a system is modelled as a whole, it is often difficult to comprehend the component parts due to the vast number of components and the complexity of relationships. Therefore, a series of ISCM configuration abstractions have been defined which, as with other forms of abstraction, enable the composition of various layers of the system to be highlighted and the currently irrelevant parts to be masked. In order to uniquely identify and document the individual components and their combination into configurations as they are re-identified during the ISCM process, a set of attributes has been identified for which the information can be incrementally added as maintenance proceeds.

A new language, the Inverse Configuration Description Language (ICDL), has been defined to enable and describe the reclamation of legacy software system configurations. The distinct feature of this language over current methods is the emphasis is places on the interaction of the application components with their environmental counterparts and the explicit encapsulation of knowledge pertinent to a particular application domain. The ICDL may therefore be considered to be a programming-in-the-environment (PITE) language and as such is a natural extension to earlier work in the fields of programming-in-the-small (PITS) and programming-in-the-large (PITL).

The ICDL provides a convenient framework for representing software system architectures and its purposely simple command syntax facilitates the extraction and manipulation of its associated data by a number of automated tools. However, in terms of the maintainer, an alternative and more intuitive representation is also provided as a series of templates, the Proforma Increasing Complexity Series (PICS), to guide the information collection and collation process during the program comprehension phase. The structure of the ICDL and the PICS are conceptually comparable, although due to time limitations on implementation each must currently be maintained and manipulated separately. Work, however, is currently in progress to enable sharing of the underlying data and hence dispel the problems associated with maintaining multiple copies of data.

The proformas associated with the PICS exist at three levels of abstraction: generic which define rules and characteristics relevant to any software system; tailored which are adapted to a particular category of application systems within a specific domain; and specific which are only relevant to a particular application system. By systematically processing the different levels of system abstraction, information may be cumulatively added to the proformas thereby increasing their level of complexity and decreasing their level of abstraction. Ultimately a 'genetic fingerprint' of a particular application system at a particular point in time is achieved. These genetic fingerprints can be baselined and stored over a period of time to record the evolution of system as a complete configuration rather than as a collection of isolated individual components.

ISCM has parallels in the activities of the traditional SCM activities. This is particularly so in the area of software manufacture, where a viable configuration has to be built by selecting compatible components, one per version group. Although the focus of ISCM is on identifying an existent configuration rather than rebuilding a new one many of the same activities have to be considered and hence ISCM can be used a precursor to the build process. This can be attributed to the increased understanding afforded by the ISCM process during the program comprehension activity.

The activities of the model generate a number of key outputs such as the system configurations, file location maps and incrementally documented knowledge of how the system functions are implemented. These outputs cumulatively contribute to reducing the time required to understand the area of a system affected by a proposed change. Use of the ISCM process and its associated model therefore enables an 'out of control' legacy system to be gradually brought 'back under control' as if it had been developed using SCM principles from the outset. Additionally, knowledge of the system configuration and the incremental knowledge recovered during the maintenance process enables the program comprehension process to become progressively more rapid as shown in Figure C4-22.

Program comprehension is generally considered to be the key activity that underpins all subsequent maintenance and reverse engineering activities. It is also documented as being the most costly activity of the maintenance process. Hence, the adoption of the ISCM process model has the potential to realise significant savings in the cost of software maintenance process.

Chapter 5 details how semi-automated support for the completion of the PICS, manipulation of the ICDL, and provision of management support facilities are provided through the development of a meta-CASE framework, the PISCES M^4 system. Chapter 6 describes in more detail the concepts underpinning the ISCM model. This level of detail may be considered synonymous with that required for Level-3 representation of the model and for the consistent implementation of the ISCM process activities via the PISCES method. Chapter 7 describes and applies, via definition of the PISCES method, the progressive utilisation of each stage identified within the ISCM process model to a number of real-world applications.

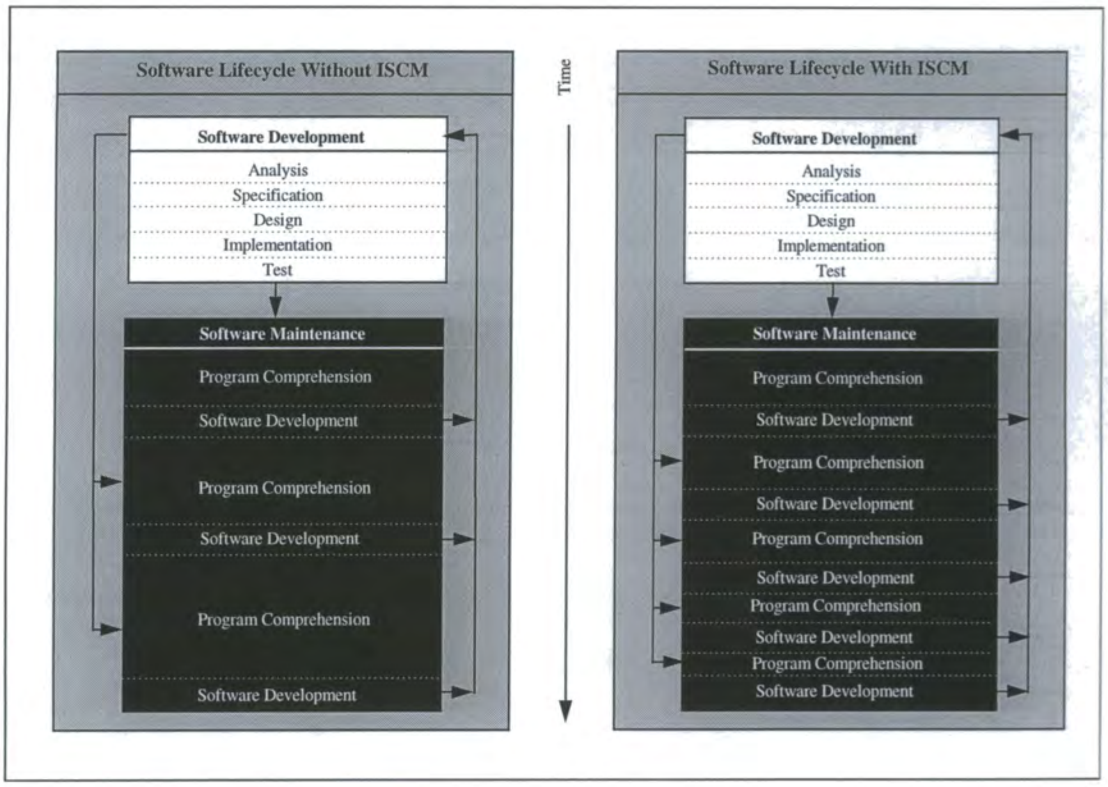


Figure C4-23 The effect of ISCM on the software maintenance process

Chapter 5

PISCES Prototype System

O, well done! I Commend your pains,
And everyone shall share i' th' gains.
And now about the cauldron sing
Like elves and fairies in a ring,
Enchanting all that you put in

Shakespeare Macbeth Act IV, Scene 1

5.1 Introduction

Chapter 4 has described the activities that underlie the ISCM process. Whilst this forms the basis of an approach for regaining control of legacy systems, there is a need to provide a degree of automated support for the process if it is to be efficiently and consistently applied; applicable to large-scale applications; and if useful output is to be generated and managed. For these reasons a series of prototype tools have been developed to enable the proformas and knowledge base on which the ISCM process is based, to be progressively constructed and the resultant information reported to the maintainer. A combination of evolutionary prototyping and incremental development techniques have been used to construct a flexible software maintenance meta-CASE (Computer Aided Software Engineering) framework into which various bespoke and host-resident tools can be linked. This chapter describes the progression of prototypes and discusses how the ideas developed in earlier chapters have been incorporated and implemented within the PISCES M^4 (MultiMedia Maintenance Manager) system framework. Future enhancements to the system are discussed in Chapter 8.

5.2 Implementation of the ISCM Process

The main purpose of the PISCES M^4 (MultiMedia Maintenance Manager) system is to facilitate the reclamation of software system configurations and to control any subsequent changes to components of these configurations during the maintenance process. To achieve this a generic framework has been developed into which a number of bespoke and proprietary tools have been and will continue to be integrated to enable the extraction, organisation, storage, querying and retrieval of information pertaining to a software system configuration. Additionally, emphasis has been placed on the provision of an environment that facilitates incremental documentation of the knowledge and understanding progressively gained during the comprehension activity of the maintenance process.

The tool has been designed to support the ISCM process described in Chapter 4 and the PISCES method detailed in Chapter 6. The key components of the PISCES M^4 system are represented diagramatically in Figure C5-1.

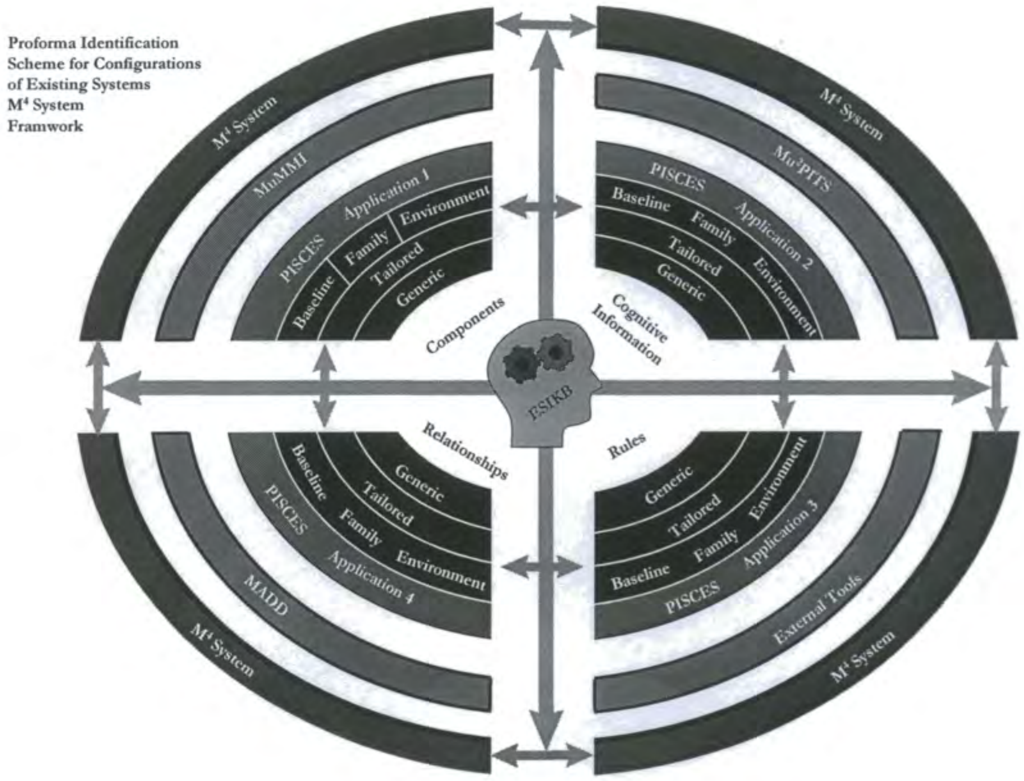


Figure C5-1 PISCES M^4 environment structure

The components may be summarised as:

- **Extensible System Information Knowledge Base (ESIB):** this forms the core of the tool. The purpose of the ESIB is to store the rules, component details and domain knowledge pertaining to a system configuration.

- **Proforma Identification Scheme for Configurations of Existing Systems (PISCES):** this tool provides support for the collection, collation and dissemination of information via the defined series of proformas described in Chapter 4. It acts as an interface between the user and the ESIKB.

- **Multimedia Application Documentation Domain (MADD):** this provides the management and control mechanisms of the tool such as access, editing, linking and storage of the configuration documentation.

- **Multimedia MultiPlatform Identification and Tracking System (Mu^{2}PITS):** this provides the mechanism for recording information about software system configurations and the individual software configuration items from which they are composed. Mu^{2}PITS also tracks the status of changes to a configuration and records the development history of components and entire configurations during their continued maintenance.

- **MultiMedia Maintenance Interface (MuMMI):** this provides an innovative means of incrementally recording and reporting information pertaining to software system configurations and the cognitive understanding gleaned during the maintenance process.

- **MultiMedia Maintenance Manager (M^4):** this provides the overall framework of the system. It binds together the ESIB, MADD, Mu^{2}PITS MuMMI, and PISCES tools and enables external point function tools such as those for version management and information extraction to be linked into the system

The M^4 prototype was incrementally built-up through a planned and controlled series of individual tools which through their integration have gradually extended the functionality offered by the original PISCES tool. The following sections of this chapter document the developmental history of the PISCES M^4 system, discuss the features implemented in the individual tools in more detail and describe how each tool relates to the PISCES method.

5.3 PISCES Development History

A progression of prototypes have been developed in order to implement the ISCM process and PISCES method described in Chapters 4 and 6. Deliberately, these prototypes have become increasingly sophisticated, both in terms of their front-end interface and their back-end integration of different applications. This has been made possible through the exploitation of new hardware and software technology as it became available within the mainstream computing environment.

5.3.1 PISCES M^1 System

The M^1 system was the initial prototype and was essentially characterised by the features outlined in Table C5-1:

PISCES M^1 SYSTEM	
Platform:	• PC 386 • Windows 3.0
Language:	• Software Development Kit version 1.0 [274] • Microsoft C
Key features:	• Dependency information extraction • Initial implementation of Proforma Increasing Complexity Series • File location information • Incremental annotation facility • Combined browsing and partial editing
Linkage mechanism:	• Hypertext
Underlying storage:	• Flat-file for configuration description and dependency information • MS-DOS file structure for PISCES generated documentation • Host system for configuration component storage
Interface	• Text and graphics

Table C5-1 PISCES M^1 system characteristics

Example of the M¹ features are shown below in Figures C5-2 and C5-3.

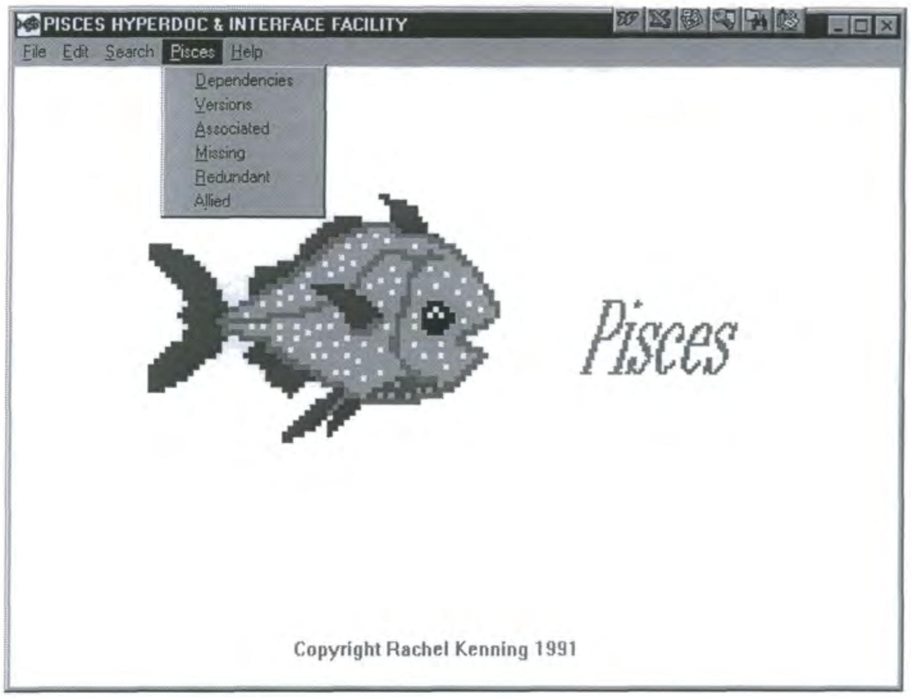
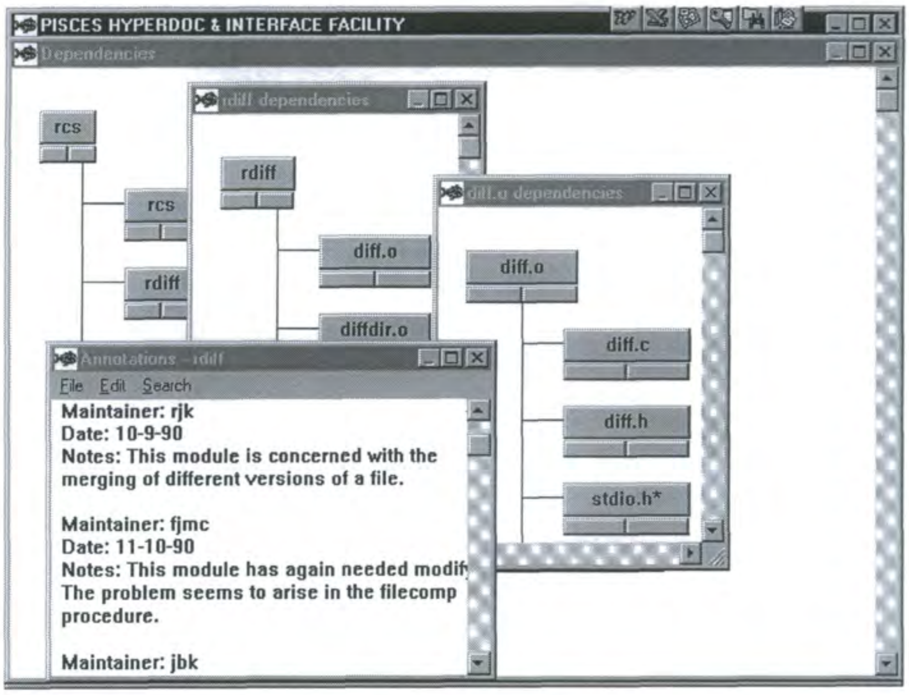

PISCES HYPERDOC & INTERFACE FACILITY
File Edit Search Pisces Help
Dependencies
Versions
Associated
Missing
Redundant
Allied
Pisces
Copyright Rachel Kenning 1991

Figure C5-2 PISCES M¹ command interface



Maintainer: rjk
Date: 10-9-90
Notes: This module is concerned with the merging of different versions of a file.

Maintainer: fjmc
Date: 11-10-90
Notes: This module has again needed modif
The problem seems to arise in the filecomp procedure.

Maintainer: jbk

Figure C5-3 PISCES M¹ dependency maps & annotation facility

5.3.2 PISCES M^2 System

The M^2 system incorporated a more sophisticated back-end to the PISCES toolset, the MultiMedia Application Documentation Domain (MADD). Additionally, it provided an innovative approach to the representation of maintenance information through the MultiMedia Maintenance Interface (MuMMI). Compatibility with the M^1 system was provided through an OLE link to the original tool. The key features of the M^2 system are summarised in Table C5-2.

PISCES M^2 SYSTEM	
Platform:	• PC 486/Pentium • Windows 3.1 and 95
Language:	• Visual Basic version 3.0
Key features:	• Integration with PISCES M^1 tool • Innovative multimedia maintenance interface for display and incremental documentation (MuMMI) • Facilities for management and control of all file types (MADD) • Natural language control facility • Separate browsing and editing levels • Password protection between browsing & editing levels • Rapid response times
Linkage mechanism:	• Hypermedia at document level of granularity
Underlying storage:	• *Makefile* for configuration description and dependency information • Windows file structure for PISCES generated documentation • Host system for configuration component storage
Interface	• Text, graphics, audio, video, animation

Table C5-2 PISCES M^2 system characteristics

The tool interface is represented below in Figures C5-4 and C5-5.

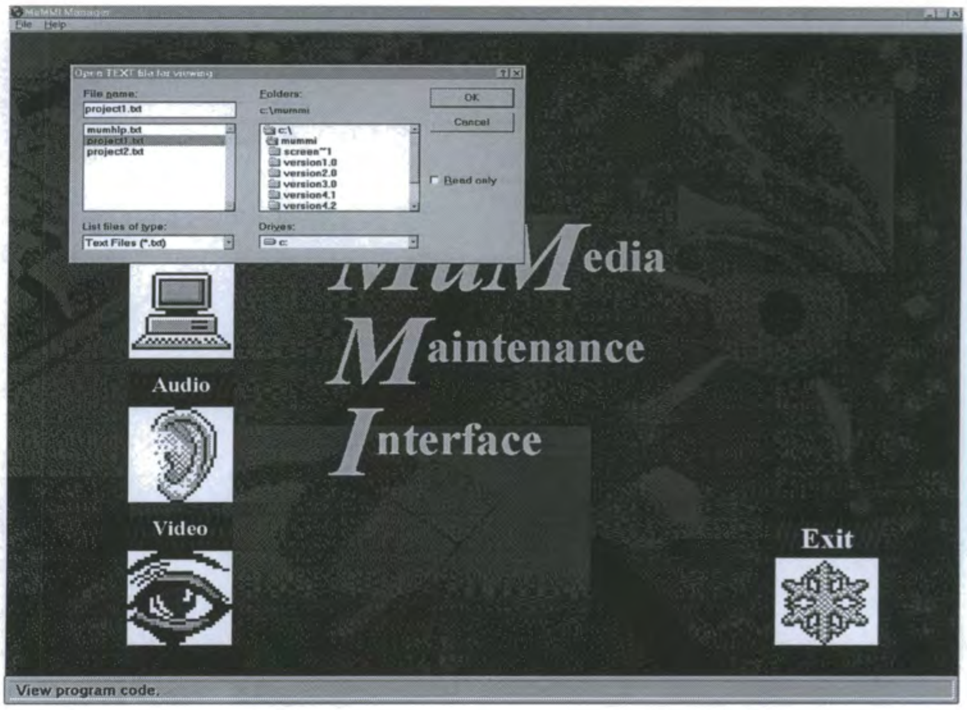
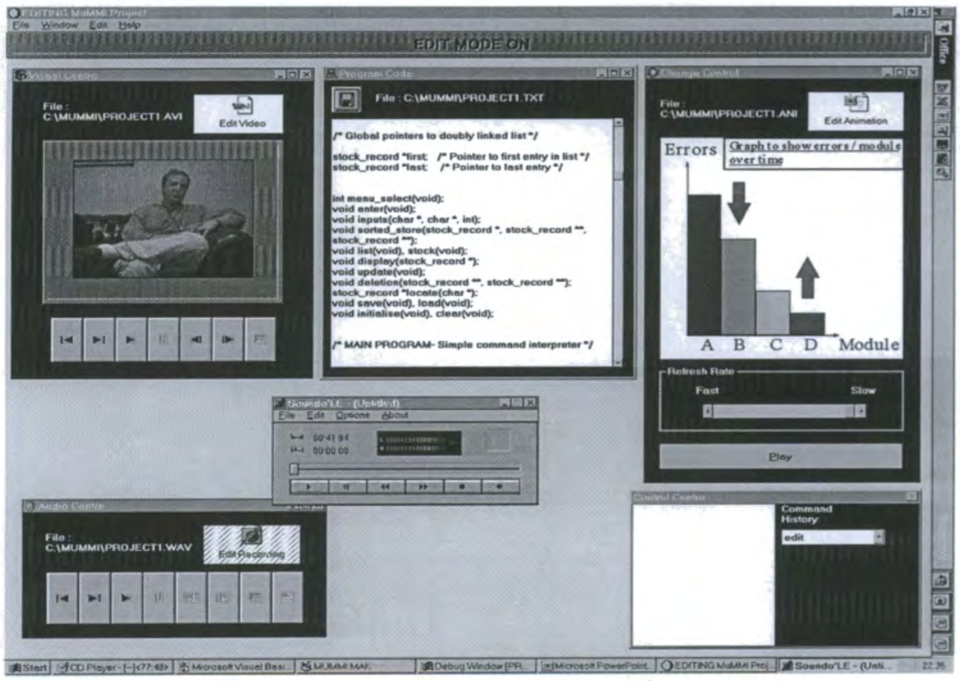


Figure C5-4 graphical menu and project selection window



```
/* Global pointers to doubly linked list */

stock_record *first;   /* Pointer to first entry in list */
stock_record *last;    /* Pointer to last entry */


int menu_select(void);
void enter(void);
void inputs(char *, char *, int);
void sorted_store(stock_record *, stock_record **,
stock_record **);
void list(void), stock(void);
void display(stock_record *);
void update(void);
void deletion(stock_record **, stock_record **);
stock_record *locate(char *);
void save(void), load(void);
void initialise(void), clear(void);


/* MAIN PROGRAM- Simple command interpreter */
```

Figure C5-5 display of multimedia file types & activation of video editor

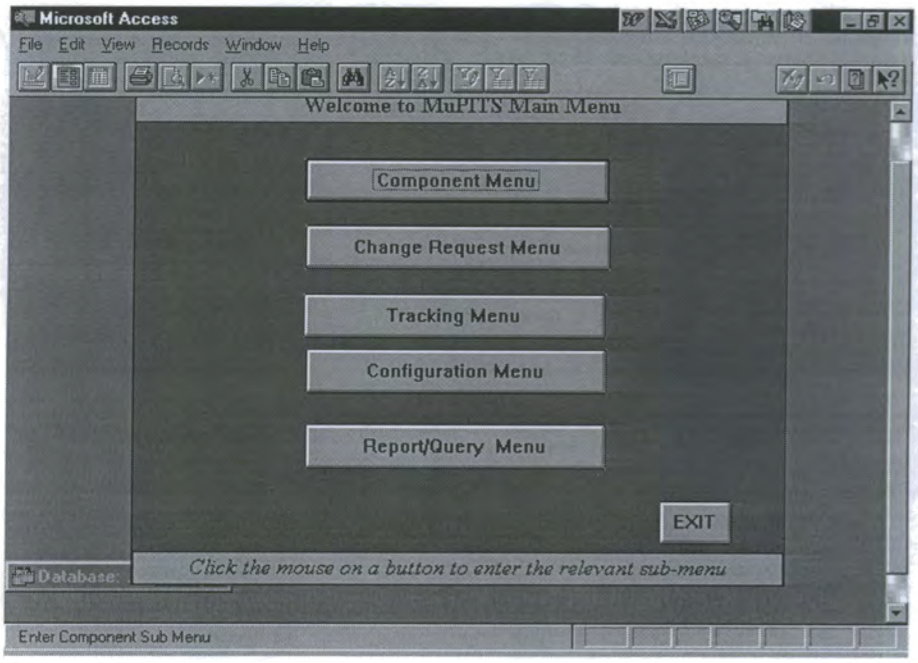
5.3.3 PISCES M^3 System

The PISCES M^3 system incorporated the Mu^{2}PITS system to the PISCES framework. This had the effect of adding facilities for documenting and tracking changes to the reclaimed software system configurations and their respective components. The system also provided the facility for documenting information regarding specific components and their combination into viable configuration listings. The key features of the M^3 system are summarised in Table C5-3.

PISCES M^3 SYSTEM	
Platform:	• PC Pentium • Windows 3.1 and 95
Language:	• Microsoft Access version 2.0 • Access Basic • Visual Basic
Key features:	• Integration with PISCES M^1 and M^2 tools • Multimedia and multi-platform configuration and component identification • Change tracking and documentation • Status accounting facilities • Comprehensive reporting mechanisms • Password protection
Linkage mechanism:	• Relational links between configuration components • Hypermedia links for documents
Underlying storage:	• *Makefile* for configuration description and dependency information • Relational database for relationship and dependency information • Windows file structure for PISCES generated documentation • Host system for configuration component storage
Interface	• Forms based interface for configuration and change tracking • Text, graphics, audio, video, animation for documentation

Table C5-3 PISCES M^3 system characteristics

Examples of the M³ system facilities are shown in Figures C5-6 and C5-7.



Click the mouse on a button to enter the relevant sub-menu

Figure C5-6 Mu²PITS main menu

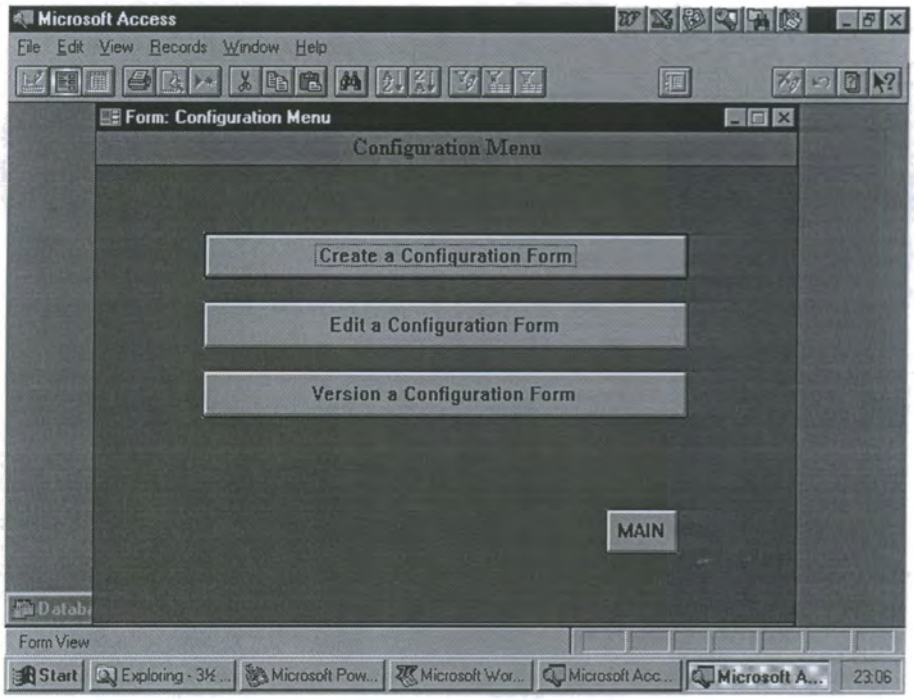


Figure C5-7 Process configuration form menu

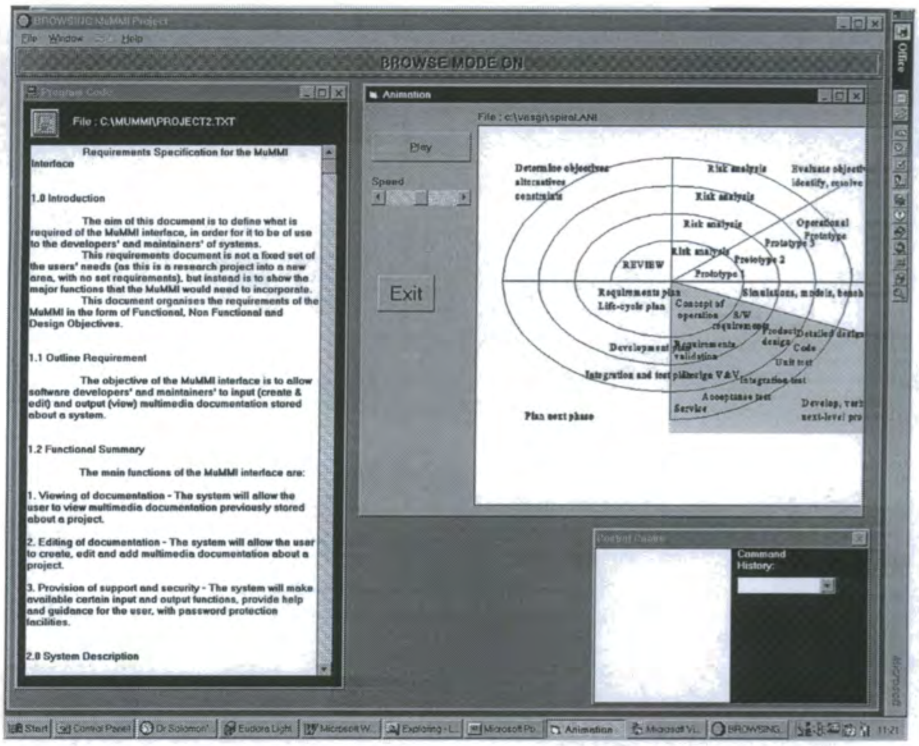
5.3.4 PISCES M^4 System

The PISCES M^4 system represents the state of the integrated system at the current time, although new features are continually being added as development proceeds. Essentially the M^4 system provides a more seamless integration of the different tools outlined so far and enables a more comprehensive set of links into other program dependency tools to be made. The extended facilities of the M^4 systems are summarised in Table C5-4:

PISCES M^4 SYSTEM	
Platform:	• PC Pentium • Windows 95 and NT
Language:	• Microsoft Access version 2.0 • Microsoft Office 97 [255] • Access Basic • Visual Basic
Key features:	• Seamless integration with PISCES M^1, M^2 and M^3 systems • Additional tool hooks to external environments, for example version control facility. • Finer granularity hypermedia linkage within textual documents • Limited media to media pinpoint hypermedia linkage. • Updated proforma series including hypertext links to multimedia file types and optional generation of HTML files. • Lifecycle process model support
Linkage mechanism:	• Relational links between configuration components • Hypermedia links for documents at within document level for text and graphics
Underlying storage:	• As M^3 system
Interface	• Seamless interface to all tools

Table C5-4 PISCES M^4 system characteristics

Examples of the M⁴ system facilities are shown in Figures C5-8 and C5-9.



File : C:\MUMMI\PROJECT2.TXT

Requirements Specification for the MuMMI Interface

1.0 Introduction

The aim of this document is to define what is required of the MuMMI interface, in order for it to be of use to the developers' and maintainers' of systems.

This requirements document is not a fixed set of the users' needs (as this is a research project into a new area, with no set requirements), but instead is to show the major functions that the MuMMI would need to incorporate.

This document organises the requirements of the MuMMI in the form of Functional, Non Functional and Design Objectives.

1.1 Outline Requirement

The objective of the MuMMI interface is to allow software developers' and maintainers' to input (create & edit) and output (view) multimedia documentation stored about a system.

1.2 Functional Summary

The main functions of the MuMMI interface are:

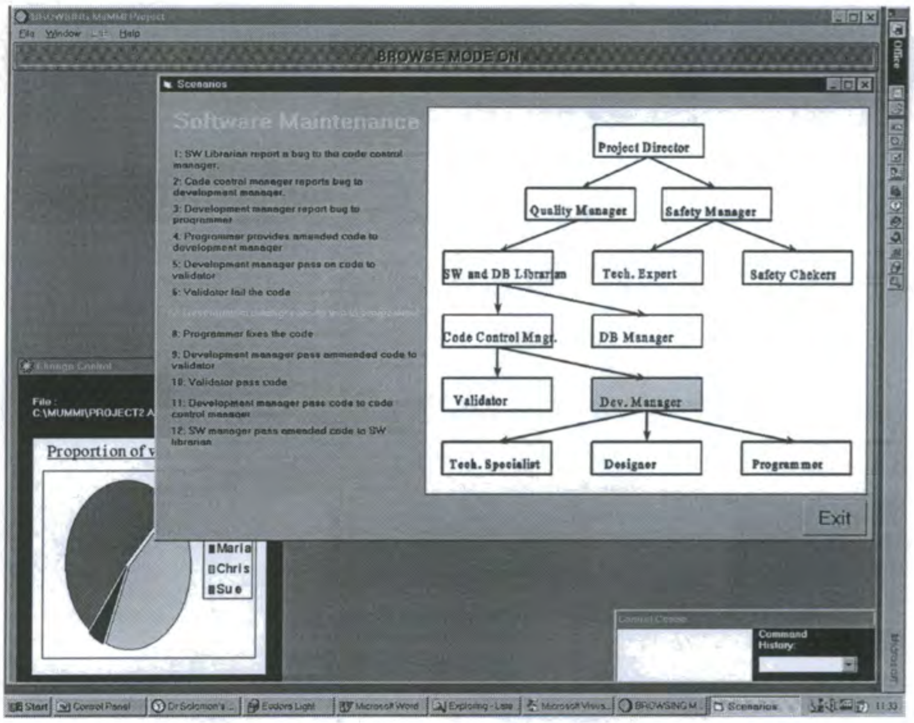
1. Viewing of documentation - The system will allow the user to view multimedia documentation previously stored about a project.

2. Editing of documentation - The system will allow the user to create, edit and add multimedia documentation about a project.

3. Provision of support and security - The system will make available certain input and output functions, provide help and guidance for the user, with password protection facilities.

2.0 System Description

Figure C5-8 M⁴ system MuMMI requirements document & lifecycle approach



1: SW Librarian report a bug to the code control manager.

2: Code control manager reports bug to development manager.

3: Development manager report bug to programmer

4: Programmer provides amended code to development manager

5: Development manager pass on code to validator

6: Validator fail the code

[illegible]

8: Programmer fixes the code

9: Development manager pass ammended code to validator

10: Validator pass code

11: Development manager pass code to code control manager

12: SW manager pass amended code to SW librarian

Figure C5-9 Animated company maintenance procedures

5.4 The PISCES M^4 System Description

As a result of the research and development of the ISCM process and PISCES method a flexible meta-CASE framework has been developed rather than a rigidly defined more traditional CASE tool. This approach to the provision of semi-automated support has a number of advantages:

- ***Control***: an underlying core set of facilities can be integrated into the toolset to satisfy the requirement to regain and keep control of a system in a defined and consistent manner.

- ***Adaptability***: the framework enables the integration of different tools into the toolset to support the individual working practices and methods of different maintenance organisations.

- ***Extensibility***: information gathered from other sources or tools can be brought into the framework and maintained as a coherent set of data.

- ***Evolution***: new technologies can be exploited as they come into the mainstream computing community.

- ***Feedback***: the incremental prototyping approach enables an early working product to be developed whilst still providing the opportunity to encompass the resultant feedback into development of later parts of the overall system.

This section describes in more detail the functions provided by each of the key components currently integrated within the M^4 system prototype.

5.4.1 PISCES

The PISCES (Proforma Identification Scheme for Configurations of Existing Systems) tool encompasses both the earliest and most recent features of the M^4 system. The activities of the PISCES tool are primarily concerned with the extraction and recording of information regarding the components and configuration of an application and the overall environment in which they play a part. As such the tool primarily implements the PICS described in Chapter 4, although close linkages are also formed with the information recorded by the Mu^{2}PITS system (see section 5.4.3) and with any host system tools that aid the extraction of information about a system's components. The following facilities are offered by the PISCES tool [264, 270]:

- ***Implementation of the PICS***: the templates associated with the proforma increasing complexity series (PICS) are provided on-line as a guide to the consistent information collection and documentation of any software system being maintained. Within the latest prototype extensive use has been made of the facilities offered by the Microsoft Office 97 suite of programmes, and in particular the use of Word 97 [255] to implement the PICS templates. As maintenance occurs, information may be successively added and saved using one of two approaches: the PICS may be updated and saved in the normal manner or a baseline PICS may be struck and the template saved as a new version. The point at which baselines are struck is definable by the maintenance organisation, for example, this may be on a temporal basis or on a per maintenance basis after a change or series of changes have been made. Progressive baselining of the PICS enables an evolutionary history of an entire system to be recorded.

- ***Creation of HTML links and web pages***: by creating the PICS within the Microsoft Office 97 suite of programs, the facility to add HTML links and covert the PICS to web pages is enabled. Maintenance across the web is however outside of the scope of this project and hence this facility has not yet been fully exploited, although it is the intention to do so in future versions of the tool.

- ***Dependency information extraction and viewing***: linkage to tools such as awk, grep and Make for extraction of dependency information; access to the resultant components such as Makefiles; and study of the code when combined with the maintainers expertise enables dependency information to be recovered and documented. Additionally small bespoke utilities enable features such as the resultant dependency tree structures to be displayed.

- ***File location information and maps***: the physical file locations for components can be identified and displayed through the use of simple data extraction utilities on either a per component or per system basis.

- ***Incremental annotation facility***: provision of a simple editor enables notes to be made regarding the understanding of the system gained during the comprehension process, or as a means of providing a temporary note to the maintainer, for example, to flag an activity that still needs to be carried out or to warn other maintainers of a troublesome section of code etc.

5.4.2 MADD

The MADD (Multimedia Application Documentation Domain) environment supports the management and control of, and access to the underlying multimedia types. A 'natural-language' control centre provides the primary linkage mechanism within the environment enabling links at a component to component level of granularity to be established between the different project documents. The key facilities provided by the MADD environment are [128, 268, 270]:

- *Viewing and loading of stored documentation:* this facility enables the multimedia documentation previously stored on a project within the M^4 system to be viewed. This includes the ability of the M^4 system to display recorded video, to play audio, to display 'pure' text or texts with incorporated animation or graphics, and to display statistics in a graphical or animated format. The system also enables concurrent viewing of different multimedia attributes, for example, it allows code to be viewed alongside an associated video recording.

- *Creation and saving of the documentation:* this facility enables new multimedia documentation to be input into the M^4 system, stored within the M^4 environment as a project directory and subsequently accessed through the MuMMI interface. These files can be in the form of pre-recorded video and audio, graphics, animations or text.

- *Editing of stored documentation*: this facility enables changes to be made to stored documentation. This is required to keep the documentation up-to-date and concurrent with the state of the software project or simply to make corrections to erroneous documentation. In the case of audio and video files, these may be externally edited and replaced in the system, or the editing tools may be loaded and the file edited through the M^4 system itself. Text documentation may be displayed as 'pure' text or may be associated with other multimedia files. For example, whilst in edit mode the user can use the audio tool to record and associate a voice-over with a piece of text, thereby adding value to and aiding the understanding of an otherwise less descriptive pure text file.

- *Appending to stored documentation:* this facility enables new information to be appended to the existing versions of the documentation held within the M^4 system. This may be required after the advancement of the documented software project, with the aim of keeping the documentation up-to-date. Although closely allied to the editing function, implementation is handled differently, as changes are sequential in nature thereby building up a change history of a particular component. The creation of new files and the formation of linkages between them and their predecessors can handle this feature.

- ***Deletion of documentation:*** this facility enables documentation to be deleted either as complete multimedia files or as linkages to files. However, this should be supported with the disciplined practice of archiving the documentation onto some form of back-up medium such as a tape streamer before allowing it to be deleted from the project environment.

- ***Provision of security and access rights***: this facility provides a security mechanism in the form of password access. This is important because only certain personnel within an organisation may have the rights to change project documentation. This facility is supported through the MuMMI which has two access levels, browse and edit, with password access into the edit mode thereby limiting changes to authorised users only.

- ***Data storage and retrieval:*** this enables the multimedia documentation accessible by the M^4 system to be stored in the form of files in the user directory. There is no restriction on the organisation of these files, thereby allowing the user to adopt a preferred method of operation. This can involve the storage of independent files associated with each project in individual directories. Alternatively, storage can be via a link to a database application, as occurs in connection with the information stored within the Mu^2PITS tool (see section 5.4.3). The advantage of the latter method is the provision of a more organised framework and the ability to conduct more complex queries and searches.

It is also acknowledged that a fully configured M^4 system will store large quantities of multimedia documentation, in the form of external files. The need for a high capacity storage medium is determined by the size of video and audio files, which are in the order of megabytes for seconds of recording (the need to store minutes or hours of recording for large project needs further consideration). However this technical problem of storage is becoming less of an issue due to the development of more efficient compression methods for sound and video files such as JPEG and MPEG [37, 352], and the continuing reduction in the price of hard-disk storage. The further development of hi-density, CD-readable drives and rewriteable CDs will also facilitate cost effective storage of huge quantities of data [295].

5.4.3 Mu^{2}PITS

The Mu^{2}PITS (MultiMedia Multi-Platform Identification and Tracking System) tool supports documentation of the identifying features of system components and their integration into software configurations. The nature of the tool is such that it enables attributes such as the relationships existing between widely different file types to be recorded and the location of these components to be tracked across distributed networks. In this respect Mu^{2}PITS differs from many of the traditional configuration management tools which tend to concentrate on text based products. Mu^{2}PITS also supports the production of change requests and tracking of the status of changes throughout the maintenance process. The key facilities provided by the Mu^{2}PITS tool are [125, 270]:

- ***Documentation of component attributes***: this facility enables information about a particular component to be documented. Once created the resultant component records may be amended, versioned or deleted. The information documented supports all types of multimedia components as well as the more traditional text-based components.

- ***Documentation of configuration composition:*** this facility enables the documented components to be associated with one or more configurations. In this way master configuration lists of the components of configurations can be built-up. Additionally, dependencies on a per component basis can be recorded. Once created the resultant configuration records may be amended, versioned or deleted.

- ***Creation/amendment/deletion of a change request:*** this facility enables a change request to be generated for a particular component. In order to avoid possible conflicts and loss of valuable data the system will allow only one active request per component at any one time. The resultant change request form may be updated if the change request has to be amended in some way after scrutiny by the change control board and is archived once the approved change has been completed or immediately if the change has been rejected.

- ***Creation/amendment/deletion of a status tracking form***: this facility enables the status of an accepted change request to be tracked during the maintenance process and it thereby allows the status of the change itself to be monitored. A status tracking form cannot be raised until the change request form has been completed and agreed. The status attribute of the tracking form will be amended during the change process to reflect the status of the component and the form will be archived once the change has been completed.

- ***Creation of reports and queries***: this facility enables information to be obtained regarding the data held within the Mu^{2}PITS database. Information may be output to a printer as reports or to the screen as queries. The reports range from listing of component details to master configuration lists to management statistics, for example details of how many components are undergoing changes at a particular point in time.

- ***Security***: the Mu^{2}PITS tool also provides a level of security in the form of password access.

5.4.4 MuMMI

The MuMMI (MultiMedia Maintenance Interface) is the front-end to the M^4 system, and is used to display the multimedia documentation [128, 267, 268, 270]. This documentation is any information of relevance to understanding the high-level design or low-level code of a software product. It is routinely added to the system throughout maintenance thereby keeping the documentation up to date in relation to the underlying software product. The interface has been designed through a 'user-centred' development approach, meeting good design practice for multimedia windows interfaces [315, 327]. In particular, the windowing environment displays a standard format in order to be consistent with other Windows based products. A Multiple Document Interface (MDI) approach was selected in order to enable the concurrent viewing of many multimedia documents.

The MuMMI co-operates closely with the MADD and is based on three levels of access:

- ***Level 1 - Hieroglyphic MuMMI Manager***: this level is used to select the particular project or system undergoing comprehension. Selection is via a graphical menu of multimedia attributes and determines whether it is the graphical, textual, audio or video documentation associated with a project that is initially displayed. Once a valid project file has been selected control is automatically switched to level-2 of the interface.

- ***Level 2 - Browsing MuMMI Project Environment***: this level is organised with respect to a particular project and is activated automatically on selection of a project file. Initially only the selected multimedia view of the project documentation and the control centre are displayed. The control centre uses a 'natural language' command interface in order to fulfil the requirements for fast operation by an experienced user group. The user communicates with the system through the entry of defined commands. A history of the commands entered within a session is maintained and can be invoked again by simple selection. As the program comprehension process proceeds the user can enter commands to display other multimedia views of the project related to the initially opened file.

The interaction between this level of the MuMMI and the MADD component of the M^4 system is one of read-only access. Thus the user is able to browse and display the multimedia documentation on accessible project files but is prevented from making any changes to stored material. The user can enter the editing levels of the M^4 system through a command in the control centre followed by access via a password protection mechanism.

- ***Level 3 - Editing MuMMI Project Environment***: the interaction between the third level of the interface and the MADD component and is that of read- and write-access. Thus at this level the user is able to edit the existing multimedia documentation and to create and store new documentation within the M^4 framework. This facility allows update or extension of the information stored on an existing product as it is recovered during the program comprehension process.

  The M^4 system makes use of external software applications for editing each of the multimedia file types. Object Linking and Embedding (OLE) connections are used to create a link between the multimedia attribute and the appropriate external application which are subsequently activated when the user selects the required edit facility. The only multimedia attribute not linked to an external application is the text window, which uses an internal editor. This is because the user does not specifically obtain a text editor with the required multimedia hardware (compared to the video, audio and graphics editors, provided in multimedia packages). However, links can also easily be generated to word processing packages such as Word for Windows in the same way as to the multimedia applications.

In addition to providing the interface to the M^4 system, the MuMMI has also addressed the issues of:

- ***Help support for users of the system***: as the M^4 system is intended for use by experienced computer personnel, help facilities are provided in the form of an on-line help system addressing the specific operations of the M^4 system rather than general computer commands.

- ***Response times:*** as already mentioned the M^4 system has been designed for use by experienced computer users and therefore requires facilities for fast operation, browsing and editing of the multimedia. The interface provides the user with short-cut keys and a 'natural' command language interface. The system also supports fast access and retrieval of multimedia documentation. However, it should be noted that the operating rate of the MuMMI system can be highly dependent on the hardware configuration.

- ***Number of users:*** the M^4 system is currently a stand-alone, single user system, although multi-user work is supported if individual sessions are separated on a temporal basis. Future development is already underway to extend the system into a fully multi-user, networked system. This would provide an environment in which long distance maintenance could take place. For example, teleconferencing techniques over the Internet could be used to link a number of MuMMI systems anywhere on the globe. This would reduce the importance of geographic location of personnel and allow experts from all areas to communicate more easily.

5.4.5 M^4 Environment

The Multimedia Maintenance Manager (M^4) system provides the overall framework of the system, binding together the ESIKB information storage mechanism, the MuMMI front-end and the MADD back-end. It also provides links into the bespoke Mu^{2}PITS and PISCES tools as well as enabling links to be made to external point function tools such as those for version management and information extraction. The evolution of the M^4 system has also incorporated animated models of the maintenance process and work is underway on a Hypermedia Browsing and Editing Facility (HyBrEF) [270] for the multimedia documentation stored within the M^4 framework. Some proposed extensions to the M^4 system are as follows:

- ***Provision of change control facilities***: with such a dynamic environment as the M^4 system and the emphasis that needs to be placed on keeping documentation concurrent with the state of the software, there is a need for rigorous change and version control. Software configuration management requirements must also extend to the ability of the M^4 system to support a family of concurrently released software product configurations to different clients. This latter requirement of recording component and product configuration details has been addressed in the Mu^{2}PITS tool. However Mu^{2}PITS does not deal with the issue of change to and subsequent storage of the actual components themselves.

As the M^4 documentation is in different multimedia formats the problems of managing changes are more complex than for text management alone particularly if the changes are to be stored as a series of deltas rather than complete files. This problem of change control may eventually be fully handled by the M^4 environment itself. Currently however, links have been provided to an external, but tailorable change and version control system for multi-platform code management [205]. This system adequately manages the versioning of textual components but does extend to managing other multimedia types. Multimedia version management within the M^4 system must therefore be handled through adherence to a set of defined procedures whilst the issue of automated multimedia version management is still being addressed [175].

- ***Hypermedia capabilities***: it is planned to extend the hypermedia capabilities of the M^4 system. Currently, full use is made of multimedia attributes, however the majority of linking and control of related items of data is implemented through the MuMMI control centre. Hypertext links existing within and between textual documents have always been implemented in the PISCES tool and have recently been extended the general M^4 environment. The next release of the M^4 system will incorporate full hypermedia extensions at both browsing and editing levels. This much finer granularity of linkage within and between the document types will enable a more elegant and targeted coupling of audio, video, graphics and text documents.

- ***Process management:*** the latest version of the M^4 system has incorporated information concerning the management of the maintenance process. This includes the use of hypertext links to explain the different stages, roles and responsibilities associated with the maintenance process and the use of animation to step through a number of process models including that of the ISCM approach.

5.4.6 ESIB

The Extensible System Information Base (ESIB) as it is currently implemented is actually an amalgamation of technologies spanning simple flat files, relational database systems and the inherent storage mechanisms of the host system. This rather piecemeal approach has arisen due to the development of individual tools which handle data storage locally but whose data can be shared through in-built data exchange mechanisms or via simple parsing tools which extract relevant data from the different toolsets as required. Whilst this approach has worked successfully and can exploit fully the advantages of flexibility there are also arguments for having a central M^4 repository, with strictly defined data structures, for use by all tools within the M^4

framework. However whilst this would offer more seamless tool integration and would lessen the likelihood of maintaining any redundant data within the M^4 system it may also make the incorporation of proprietary tools difficult or even impossible.

5.5 Application of the M^4 System to the ISCM Process

Section 5.4 has described the functionality offered by each of the main components of the M^4 system. When integrated together these components provide the underpinning framework for on-line implementation and semi-automated support for the entire ISCM process. The overall interaction of the tools within the M^4 system framework is shown in Figure C5-10 and the mapping of these tools to the salient features of the ISCM process is summarised in Table C5-5.

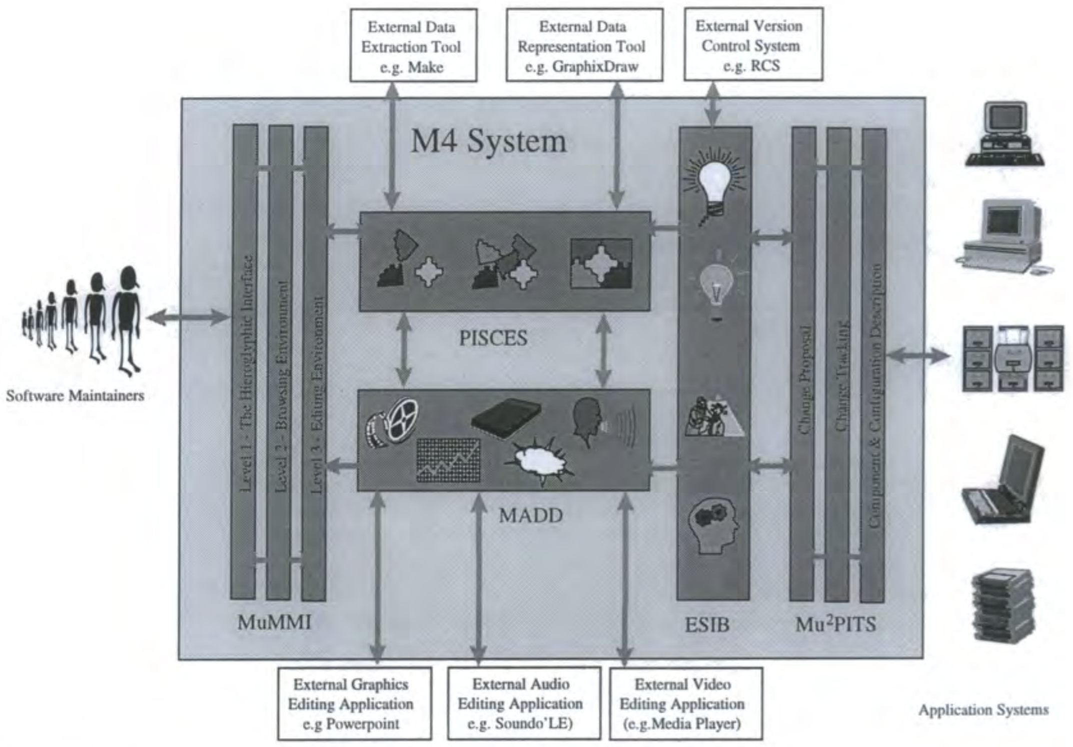

External Data Extraction Tool e.g. Make
External Data Representation Tool e.g. GraphixDraw
External Version Control System e.g. RCS
M4 System
Level 1 - The Hieroglyphic Interface
Level 2 - Browsing Environment
Level 3 - Editing Environment
PISCES
MADD
Change Proposal
Change Tracking
Component & Configuration Description
Software Maintainers
MuMMI
ESIB
Mu^{2}PITS
External Graphics Editing Application e.g Powerpoint
External Audio Editing Application e.g. Soundo'LE)
External Video Editing Application (e.g.Media Player)
Application Systems

Figure C5-10 Tool integration within the M^4 system

TOOL	ISCM PROCESS
MuMMI	• Interface to the program comprehension process • Selection of a project for maintenance • Browsing of current documentation and information regarding an application and of the application components themselves. • Links to facilities for creation and editing of multimedia documentation regarding comprehension of the application
PISCES	• Provision of generic, tailored and specific templates
Mu2PITS	• Documentation of software component details • Documentation of software configuration details • Production and update of change control proforma • Production and update of change tracking proformas
ESIB	• Storage of rules and guidelines regarding systems and their construction • Logical storage of all information pertaining to a software system configuration and its comprehension • Physical storage of information created within the M^4 environment
MADD	• Handles the creation, update and storage of information types created within the M^4 framework
M^4	• Overall ISCM environment, providing a universal interface to the activities of the ISCM process
External Editing Tools	• Physical creation and editing of the multimedia file types
External Data Collection & Representation Utilities	• Extraction of information for data representation regarding the comprehension process
External Version Control System	• Version control of text documents within the M4 framework

Table C5-5 Mapping of M^4 system features to the ISCM process activities

It is also important to establish the use of the M^4 system within the task-oriented context of the way in which maintainers performing their role. It is intended that the M^4 system is used during the program comprehension process prior to each maintenance change according to the following procedures:

- ***Browse***: initially, the maintainer would browse the program code and any associated documentation already present in the MuMMI system that appears to be of use to understanding the proposed change and the affected parts of the system.

- ***Evoke***: if appropriate, the maintainer may evoke any data extraction tools etc. existing within or integrated into the M^4 framework in order to generate additional understanding or dependency information about the construction of the system.

- ***Complete***: the maintainer should document any additional findings about the affected parts of the system within the specific proforma generated for the system under study.

- ***Update***: as the comprehension process occurs the maintainer will gain more information about the application itself, the application domain, the detailed functionality or design of the system etc. This understanding should be incrementally recorded using an appropriate multimedia type and subsequently stored within the M^4 system.

In essence the M^4 system acts as a 'maintenance oracle' capturing all previous maintenance experience within a single environment. The effects of this system are three-fold:

- ***Increased understanding***: as domain, system and application knowledge is incrementally built-up and recorded so too is the understanding of the system and its importance in the context of the organisational and system domain better understood. This has the effects of ensuring that changes can be prioritised towards those of a key business nature and it also ensures that changes can be made more safely with reduced risk of the ripple effect occurring.

- ***Faster changes***: as more information is built-up about a system the time attributed to the comprehension process should become successively reduced. This is because the prior knowledge of the maintainer themselves and of any previous maintainer is available for use and hence provides a 'head start' so that the time taken to understand a change and its impact on the rest of the system can become progressively more rapid.

- ***Foundation for re-engineering***: the increased documentation and understanding of a system, as well as notes about a system or collection of metrics about errors etc. assists with identifying the areas of the system that will most benefit from being re-engineered. Additionally, identification of the components and records of how they are synthesised into complete configurations provides a solid foundation on which the actual re-engineering process can be based.

Additionally, although the use of the M^4 system has primarily being discussed within the context of its use during maintenance, maximum benefit of the M^4 system comes as a result of it being used from the inception of a green-field project. In these circumstances, the M^4 system can be populated with the complete set of components and documentation about an application. Additionally, all design decisions, key meetings or reasoning of an informal, semi-formal and formal nature can be captured from the outset of development through a combination of the different multimedia attributes. Used in this way, the M^4 system pre-empts the maintenance process and acts as a 'maintenance escort' during hand-over of the application.

5.6 Summary

This chapter has described the development, features and structure of the PISCES M^4 system. The M^4 system is an innovative and flexible meta-CASE environment that provides on-line implementation and semi-automation of each stage of the ISCM process. The M^4 system provides the framework into which other tools of a bespoke or proprietary nature can be incorporated to enable the understanding and configuration of a software system to be documented using a more heterogeneous and enriched mix of information types than has previously been possible.

The M^4 system framework may be summarised as having a multimedia (MuMMI) front-end, an underlying repository (ESIB) and a control and management back-end (MADD). In addition to this there is a data collection and collation mechanism (PISCES), support for the SCM activities of identification, control, status accounting and audit (Mu^{2}PITS), and the integration of other tools such as those for data extraction and version control. Together these tools span the entire range of the ISCM process activities and through their defined usage during development and maintenance plus their ability to link to other tools may be considered to provide a complete maintenance environment

The support provided by facilities of the M^4 system and its close co-operation with the practical application of the PISCES method will be discussed in more detail in Chapter 6 in order to illustrate its practical applicability to the program comprehension process and maintenance of a number of laboratory and 'real-world' programs.

Chapter 6

Application

And go we, lords, to put in practice that
Which each to other hath so strongly sworn

Shakespeare Love's Labours Lost, Act I, Scene II

6.1 Introduction

This chapter describes the use of the ISCM model, PISCES method and PISCES M^4 system for reclaiming and incrementally documenting the configurations of legacy software systems during the maintenance process. Due to the nature of the program comprehension activity, which necessitates a high degree of human reasoning and inference, the ISCM process cannot, as yet, be fully automated. Therefore within the scope of this work a combination of maintainer expertise and manual procedures have been used alongside the semi-automated activities provided by the various tools integrated within the PISCES M^4 system. Suggestions for how increased automation of the ISCM process might be achieved in the future are discussed in Chapter 8.

Modelling of the applications has been an iterative process throughout the study rather than a single evaluative step at the conclusion of the project. This is in keeping with the user-centred approach to systems development, which realises the importance of task-oriented development and early and continuous user-interaction. This is especially true in connection with the development of the multimedia interface to the PISCES M^4 system which to be truly effective must be treated as an integral part of the system rather than a 'bolted-on' afterthought. Indeed, in contrast to many systems, the MuMMI interface forms the central part of the system through which the individual tools are subsequently added.

The iterative nature of the application modelling process enabled feedback to be made into the ISCM model and PISCES method. This decreased the risk of finding significant flaws in the full model once it was developed. Additionally, such an approach proved to be highly effective, particularly in connection with requirements capture for the individual components of the PISCES M^4 system toolset.

To demonstrate the capabilities of the ISCM model and PISCES method, this chapter uses a number of application systems to progressively walkthrough the stages of the model and method, incorporating the activities, features and tools described in Chapters 4 and 5. The complete process is described for each stage of the model and at this level of detail may be considered to represent level three of the ISCM model and be synonymous with the PISCES method as alluded to earlier in Chapter 4.

6.2 Trial Levels

The effectiveness of the ISCM process was examined by modelling a number of different application programs. These ranged in size and complexity from a small calculator program with a small number of reasonably homogenous components to the PISCES M^4 system itself, which incorporated a large number of heterogeneous components. Two Unix applications, RCS (Revision Control System) and SPMS (Software Project Management System) were also modelled, since these two systems were the subject of the original manual modelling process in order to determine the initial requirements of the ISCM process and the ESIB. Within the scope of the thesis it has not been possible to model a large-scale industrial system, however the implications of modelling such systems are discussed in relation to the work undertaken and the evaluation of the method in Chapter 7.

For comparative purposes, each application is rated according to a simple scale of complexity based on:

- The number of baseline application components.
- The diversity of the component types in terms of the number of component groups having representative components.
- The complexity of the component relationships measured as a function of the intra- and inter-specific relationships
- The number of lines of application code.

The key characteristics of each of the applications studied are summarised in the remainder of this section. The rest of the chapter then discusses the effectiveness of the ISCM process, PISCES method and M^4 system in understanding and documenting these applications.

6.2.1 The Calculator : Small Scale Program

The calculator program [314] represents the smallest and least complex application that was modelled. Essentially it is a small C++ application, running under MS-DOS and compiled using the Borland C++ package version 2.0. The key complexity characteristics of the application are summarised in Table C6-1.

METRIC	RESULT
Component number	11
Component diversity	5
Intra-relationship complexity	10
Inter-relationship complexity	10
Number of lines of code	300

Table C6-1 Complexity characteristics of the calculator program

6.2.2 The Chess Game : Small-Medium Sized Program

The chess program [182] is an order of magnitude more complex than the calculator program, in terms of lines of code and the number of components associated with the application. The relationships existing between the application components are also more complex but the heterogeneity of component types is of a similar nature. Although it is also a C++ program it runs under Windows 95 and via the Visual C++ development suite of tools. The key complexity characteristics of the application are summarised in Table C6-2.

METRIC	RESULT
Component number	30
Component diversity	5
Intra-relationship complexity	20
Inter-relationship complexity	10
Number of lines of code	3,000

Table C6-2 Complexity characteristics of the chess program

6.2.3 RCS : Medium Sized Program

RCS, the Unix Revision Control System, is a sizeable but still relatively small program compared to many application systems. However, it proved to be surprisingly complex to model completely given the size of the code and the number of components even though these were fairly uniform in nature. RCS was considered to be an interesting application to model since it is in itself a configuration management facility and unique in terms of many full-scale applications in that the complete source code was available for modelling purposes. The key complexity characteristics of the application are summarised in Table C6-3.

METRIC	RESULT
Component number	60
Component diversity	10
Intra-relationship complexity	40
Inter-relationship complexity	10
Number of lines of code	15,000

Table C6-3 Complexity characteristics of the RCS application

6.2.4 SPMS : Medium - Large Sized Program

SPMS (Software Project Management System) is an application available on some but not all Unix systems. This system is considerably more complex than RCS in terms of the number of components and the number of relationships existing between them. The size of this system dictates that complete modelling of the system is not feasible, nor indeed necessary within the scope of this thesis. It thus acts as a candidate for the partial modelling and incremental documentation of only those parts of the system affected by a proposed change. However, although the number of components is high, like RCS the level of heterogeneity of the components is relatively low. The key complexity characteristics of the application are summarised in Table C6-4.

METRIC	RESULT
Component number	900
Component diversity	15
Intra-relationship complexity	90
Inter-relationship complexity	20
Number of lines of code	30,000

Table C6-4 Complexity characteristics of the SPMS application

6.2.5 PISCES M^4 system

It is often asserted that the proof of a good process, method or tool is its ability to build, document or control itself. For this reason the M^4 system was also documented and configured with respect to the ISCM process, PISCES method and M^4 system. Modelling of the M^4 system also presented some useful problems in terms of coping with the heterogeneity and complexity of interaction between the different components of the toolset and the number of third party libraries and applications on which they depend. However, to some extent, the way in which the toolset has been constructed has minimised the overall complexity of the system by ensuring that most of the relationships can be partitioned at the individual tool level and then by using standard interface protocols to pass interaction from one tool to another. The key complexity characteristics of the M^4 application are summarised in Table C6-5.

METRIC	RESULT
Component number	400
Component diversity	50
Intra-relationship complexity	40
Inter-relationship complexity	40
Number of lines of code	10,000

Table C6-5 Complexity characteristics of the M^4 system

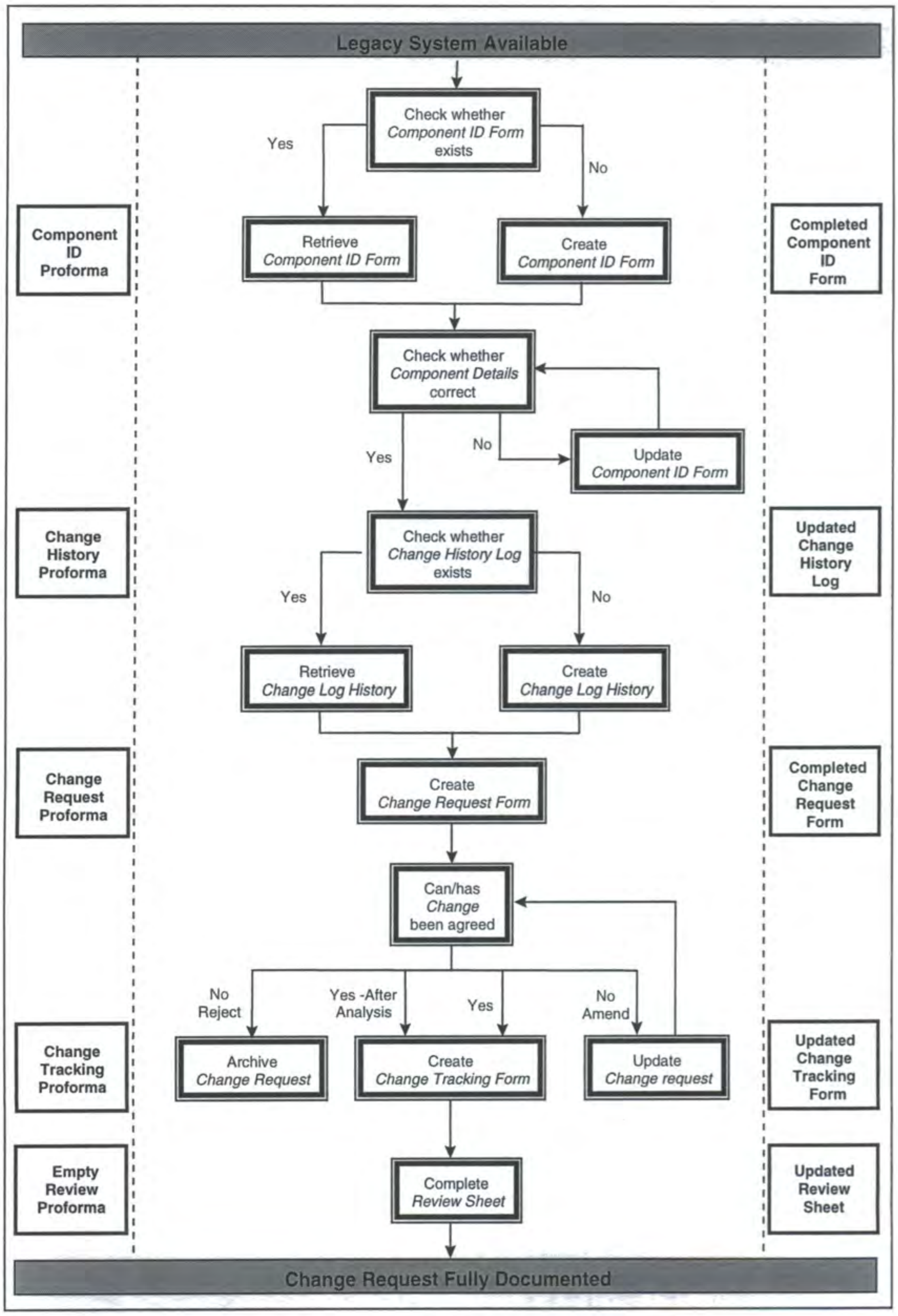
6.3 Application of ISCM process, PISCES method & M^4 system

This section describes the use of the ISCM process, PISCES method and M^4 system for maintenance of a legacy software system. Each sub-section describes the realisation of the conceptual ISCM process activities defined in Chapter 4, through the application of the guidelines and procedures established by the PISCES method described in this section, and their implementation via the M^4 system outlined in Chapter 5. The extent to which the modelling process varies according to the size and complexity of the system being maintained is also discussed.

6.3.1 Process 1 – Process Change Request

This process, initiates the maintenance of a legacy software system. Its aim is to ensure that the change request is fully documented prior to the program comprehension process commencing. The change request documents may be raised within the ISCM process itself or may be created from inside of an alternative maintenance framework and then integrated into the ISCM process.

The stages defined within the PISCES method and summarised in Figure C6-1 assume creation of the change request within the ISCM process framework.



Legacy System Available
Check whether Component ID Form exists
Yes
No
Retrieve Component ID Form
Create Component ID Form
Component ID Proforma
Completed Component ID Form
Check whether Component Details correct
Yes
No
Update Component ID Form
Check whether Change History Log exists
Yes
No
Change History Proforma
Updated Change History Log
Retrieve Change Log History
Create Change Log History
Create Change Request Form
Change Request Proforma
Completed Change Request Form
Can/has Change been agreed
No Reject
Yes -After Analysis
Yes
No Amend
Archive Change Request
Create Change Tracking Form
Update Change request
Change Tracking Proforma
Updated Change Tracking Form
Complete Review Sheet
Empty Review Proforma
Updated Review Sheet
Change Request Fully Documented

Figure C6-1 Level 3: method (work instruction) level - PISCES process change request activities

The process is essentially one of ensuring that the change required is officially documented and associated with a system component which may or may not already have a change history associated with it. Within this process all of the traditional configuration management activities are being addressed to some extent for legacy systems:

- Identification (*Creation/Retrieval/Update of* ***Component ID Form***)
- Control (*Raising/Agreeing of* ***Change Proposal*** *and* ***Change Tracking Forms***)
- Status Accounting (*Creation/Retrieval/Update of* ***Change Log***)
- Audit (*Completion of* ***Review Form***)

The tool support is provided for each of these activities by the *M^4 system.* In particular the **Component ID**, **Change History Log**, **Change Request** and **Change Tracking** forms are handled by the *Mu^2PITS* tool. The **Review Sheet** is maintained as a *Word* document.

In connection with this process, the size and complexity of the applications studied were found to have no significant effect on the effort required for documenting the change request. This is not surprising since the nature of the process is one of 'intention' rather than 'analysis' at this preliminary stage.

6.3.2 Process 2 – Conduct Analysis of Legacy System

This process, involves the initial analysis of either an entire legacy system or more commonly an analysis of only those parts of the system affected by the proposed change. The aim of the process is to identify and document the relevant component parts of the system, the relationships existing between these components and the relationships between the components and their environment. The process involves querying of the ESIKB for any previously recorded knowledge about the system being maintained; determining whether any relevant proformas exist at the varying levels of abstraction; production or update of the proformas and production of configuration and dependency lists. The process also ensures that any knowledge reclaimed during this initial analysis phase is incrementally recorded for use by future maintainers. The PISCES method stages associated with this stage of the ISCM process are summarised in Figure C6-2.

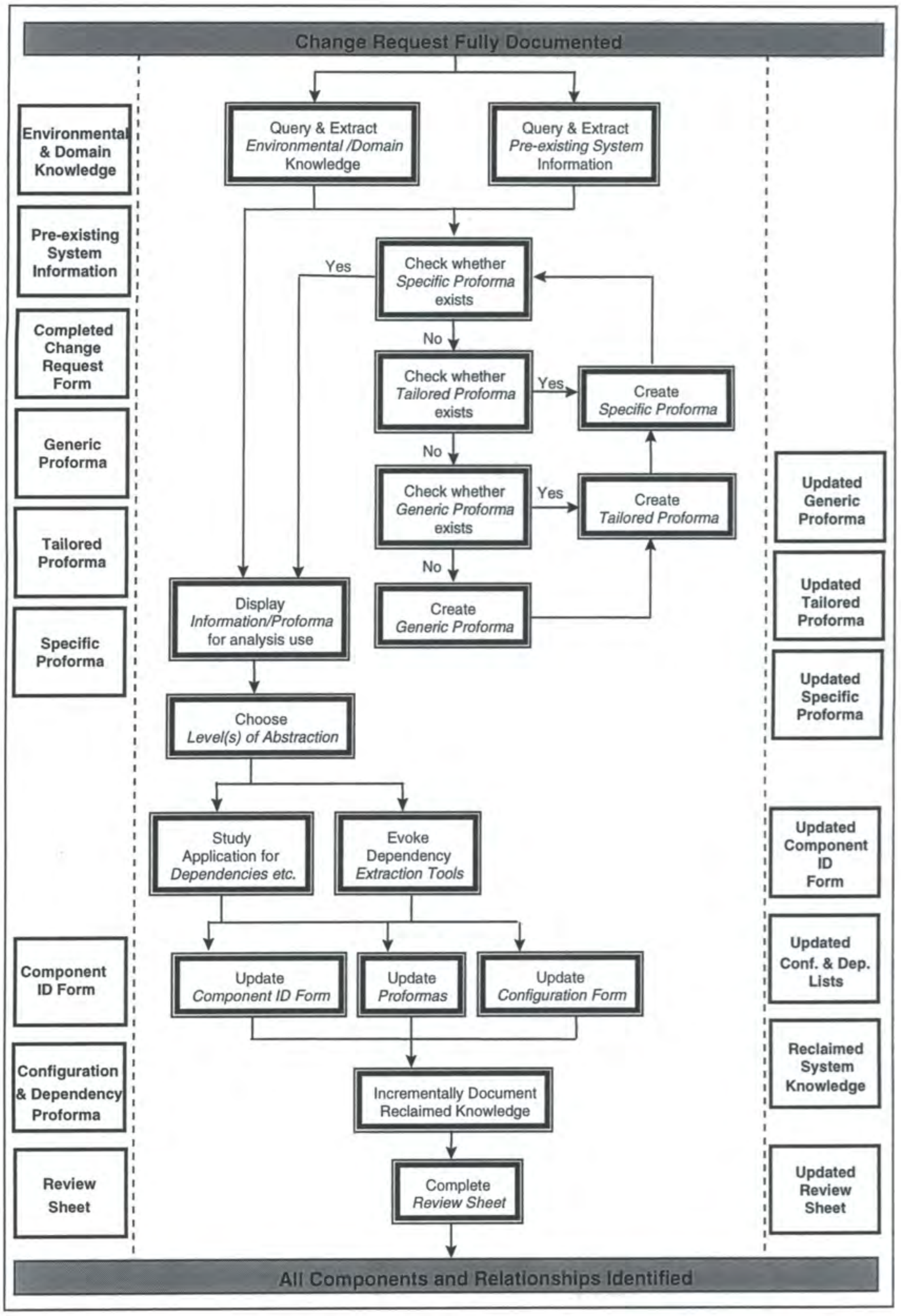

Change Request Fully Documented
Environmental & Domain Knowledge
Pre-existing System Information
Completed Change Request Form
Generic Proforma
Tailored Proforma
Specific Proforma
Component ID Form
Configuration & Dependency Proforma
Review Sheet
Query & Extract Environmental /Domain Knowledge
Query & Extract Pre-existing System Information
Check whether Specific Proforma exists
Yes
No
Check whether Tailored Proforma exists
Yes
Create Specific Proforma
No
Check whether Generic Proforma exists
Yes
Create Tailored Proforma
No
Create Generic Proforma
Display Information/Proforma for analysis use
Choose Level(s) of Abstraction
Study Application for Dependencies etc.
Evoke Dependency Extraction Tools
Update Component ID Form
Update Proformas
Update Configuration Form
Incrementally Document Reclaimed Knowledge
Complete Review Sheet
Updated Generic Proforma
Updated Tailored Proforma
Updated Specific Proforma
Updated Component ID Form
Updated Conf. & Dep. Lists
Reclaimed System Knowledge
Updated Review Sheet
All Components and Relationships Identified

Figure C6-2 Level 3: method (work instruction) level - PISCES conduct analysis of legacy system activities

The process essentially conducts a preliminary analysis of the system undergoing maintenance. All relevant information is extracted from the underlying ESIB and made available to the maintainer. Any updates regarding the system components or their relationships based on the level of abstraction selected are undertaken. This process is primarily concerned with the identification activity of the SCM process as applied to legacy systems. However, the audit trail of ISCM activities is maintained throughout the process:

- Identification (*creation/update of* ***Proformas, Component ID Forms, Configuration Lists*** *and* ***Reclaimed Knowledge****)*
- Audit (*completion of* ***Review Sheet***)

Again tool support is provided for each of these activities by the *M^4* system. In particular the creation/update of the **Proformas (PICS)** is handled by the *PISCES* tool, **Component ID Forms** and **Configuration Lists** by the *Mu^2PITS* tool, incremental documentation of the **Reclaimed Knowledge** by the *MuMMI* and *MADD* environments and as before the **Review Sheet** is maintained as a *Word* document accessible through the general *M^4* framework.

In connection with this process, the size and complexity of the applications studied impacted in a number of ways on the effort required for analysing the system. Firstly, a distinction had to be made between those systems which can be studied in their entirety and those systems for which only the affected parts of the system can be feasibility studied each time a maintenance change is required. Whilst the creation of the generic and tailored proformas were not affected by the size or complexity of the applications, production of the specific proforma was found to be significantly affected due to the increased amount of information that needs to be analysed and subsequently entered. Similarly, the identification of dependency information was more time consuming for larger applications and particularly for those with a high degree of interaction with environmental components. However, the actual method of documenting the dependencies once identified was no more complex for large heterogeneous systems than for small homogenous systems. Recording of the knowledge once reclaimed about a system was also not affected by the size of the system.

6.3.3 Process 3 – Populate Extensible System Information Knowledge Base

This process may essentially be regarded as a continuation of the analysis process. However, whilst the analysis process is concerned with collecting information pertaining to a system, the population process is responsible for ensuring that all the information reclaimed and documented about the system is stored within the ESIB such that it is available for future reference.

As mentioned in Chapter 5, the ESIB is currently composed of a number of different data storage mechanisms and record types brought together within the M^4 framework. Whilst this offers flexibility in terms of tool integration and incorporation of custom working practices, it can lead to data redundancy and it necessitates careful handling of the storage, editing and versioning associated with each information type. The population process therefore defines the way in which information should be progressively stored within the ESIB.

The relationship between the *'populate ESIB'* process and the *'conduct analysis process'* described in the previous section is actually one of iteration rather than one limited to succession. The iterative cycle may either fall within a single period of maintenance or may occur over consecutive maintenance changes. This is due to the incremental nature of the reclamation process that progressively recovers and stores information about a system via one or more of the supported media types or using the supplied proformas. Over time this augmentation of the material existing in the ESIB makes possible a more highly evolved start to the program comprehension process. The PISCES method stages associated with this stage of the ISCM process are summarised in Figure C6-3.

This process essentially controls the saving, within the ESIB, of all components browsed, created or updated as part of the analysis process. The PISCES method activities are intended to act as a guide encompassing all component types rather than being a mandatory set of instructions. This is to maintain flexibility of approach both in terms of the working practices of maintenance organisations and with regard to the mechanisms by which the different tools incorporated within the M^4 framework handle the storage of components. Detailed level instructions can be found in the user-guides or on-line help systems associated with each tool. As this part of the process is restricted to the update and saving of information it is primarily concerned with applying the SCM change control activity to program comprehension of legacy systems. Again an audit trail of the ISCM activities conducted is maintained:

- Change control (controlling updates and creation of **Proformas, Component ID Forms, Configuration Lists, Reclaimed Cognitive & Domain Knowledge, Existing System Information**)
- Audit (completion of **Review Sheet**)

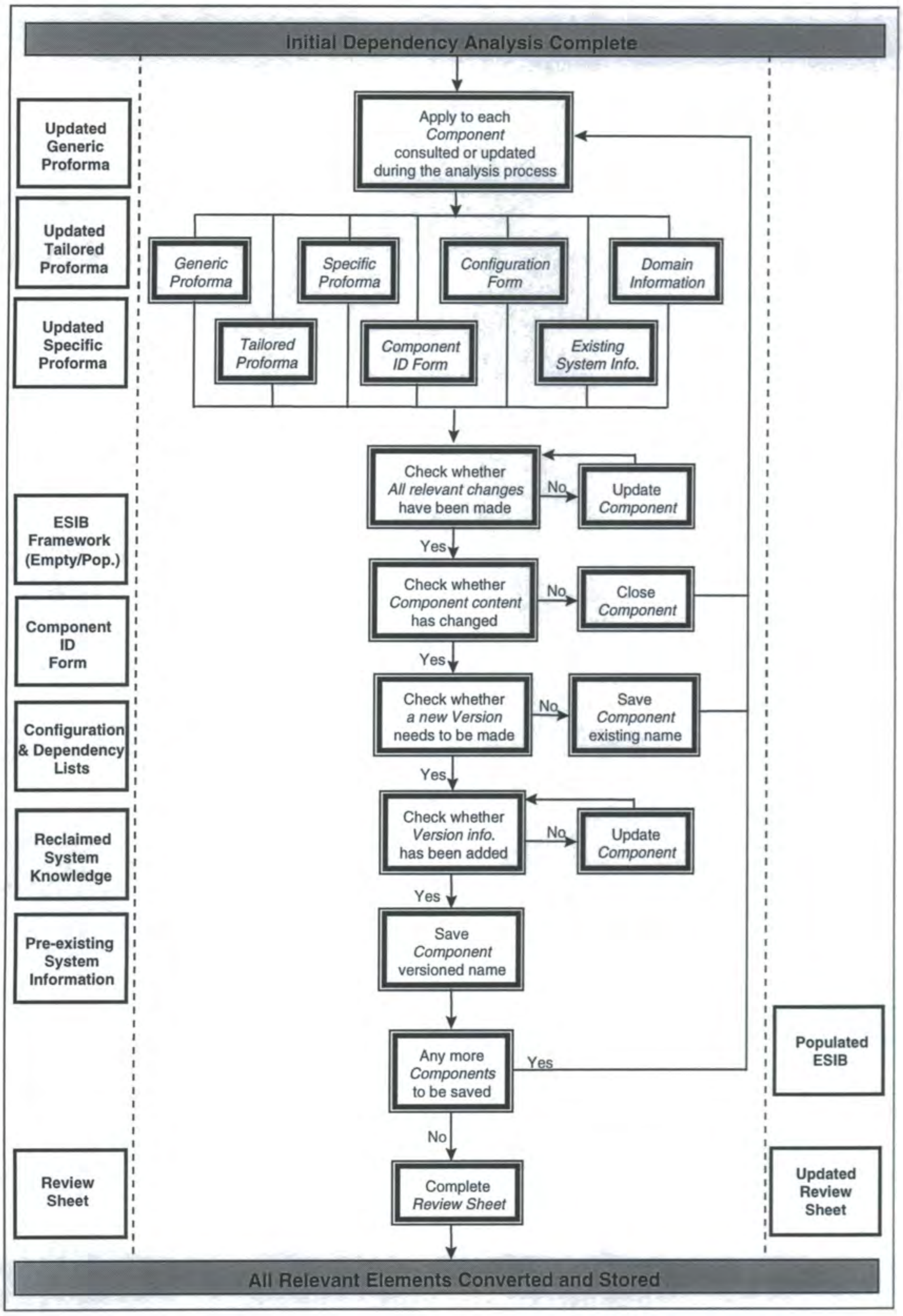

Initial Dependency Analysis Complete
Updated Generic Proforma
Updated Tailored Proforma
Updated Specific Proforma
Apply to each Component consulted or updated during the analysis process
Generic Proforma
Specific Proforma
Configuration Form
Domain Information
Tailored Proforma
Component ID Form
Existing System Info.
Check whether All relevant changes have been made
No
Update Component
Yes
Check whether Component content has changed
No
Close Component
Yes
Check whether a new Version needs to be made
No
Save Component existing name
Yes
Check whether Version info. has been added
No
Update Component
Yes
Save Component versioned name
Any more Components to be saved
Yes
No
Complete Review Sheet
ESIB Framework (Empty/Pop.)
Component ID Form
Configuration & Dependency Lists
Reclaimed System Knowledge
Pre-existing System Information
Review Sheet
Populated ESIB
Updated Review Sheet
All Relevant Elements Converted and Stored

Figure C6-3 Level 3 : method (work ins.) level - PISCES populate system information knowledge base activities

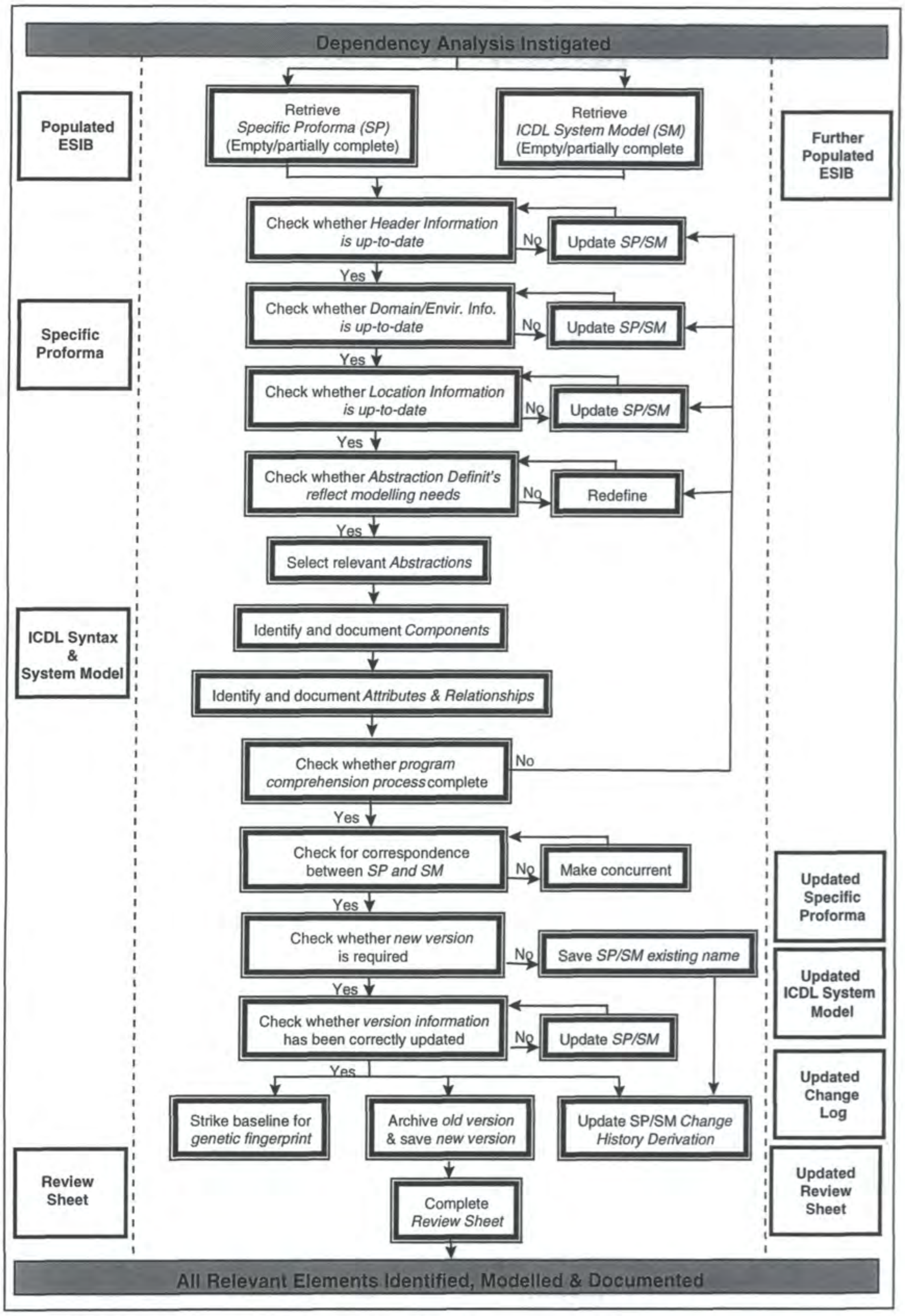
Tool support is provided for each of these activities via the storage mechanism of the individual tools within M^4 system. In particular storage of the **Proformas** and **Review Sheet** is handled by *Word*, the **Component ID Forms** and **Configuration Lists** by the *Mu²PITS* tool, and the

Reclaimed Environmental & Domain Knowledge and **Existing System Information** by the *MADD*. Additional control and protection with respect to which changes can be made and by whom is provided through a password access mechanism between the browsing and editing levels of the *MuMMI* and similarly through password access in the *Mu^{2}PITS* tool.

In connection with this process, the size and complexity of the applications studied were found to have no significant effect on the effort required for documenting the change request. This is not surprising since the nature of the process is one of 'storage' rather than of 'analysis and recovery' which is done in the previous stage.

6.3.4 Process 4 – Produce natural representation of ISCM model representation

This process describes in more detail how software system configurations can be modelled within the ISCM process using the PISCES method. The approach consists of two parallel activity strands. The first strand relates to the Proforma Increasing Complexity Series (PICS) that is a human-friendly format readily understood by customers, users and maintainers. The second strand concerns the Inverse Configuration Description Language (ICDL) system model that is a machine-oriented format that can be readily parsed by tools for automated information extraction. The process is closely tied to the *analysis process* and generally occurs concurrently with it and with the process concerning access to and retrieval of information in the ESIKB. The primary function of the PICS is to guide the retrieval and documentation of information pertinent to the system or part of the system under investigation. The process concentrates on identifying the components and relationships of a system according to varying configuration abstractions, the composition of which can be defined within the proforma and system model themselves. The resultant specific proforma or architectural model resulting at the end of the natural representation modelling phase may be considered as providing a genetic fingerprint of the state of a system at a particular point in time. Progressive recording of the variations between successive proformas (or models) over the maintenance lifetime of a product in a configuration change log enables the maintenance history of a system to be made visible. The PISCES method stages associated with this stage of the ISCM process are summarised in Figure C6-4.



Dependency Analysis Instigated
Populated ESIB
Retrieve Specific Proforma (SP) (Empty/partially complete)
Retrieve ICDL System Model (SM) (Empty/partially complete
Further Populated ESIB
Check whether Header Information is up-to-date
No
Update SP/SM
Yes
Specific Proforma
Check whether Domain/Envir. Info. is up-to-date
No
Update SP/SM
Yes
Check whether Location Information is up-to-date
No
Update SP/SM
Yes
Check whether Abstraction Definit's reflect modelling needs
No
Redefine
Yes
Select relevant Abstractions
ICDL Syntax & System Model
Identify and document Components
Identify and document Attributes & Relationships
Check whether program comprehension process complete
No
Yes
Check for correspondence between SP and SM
No
Make concurrent
Updated Specific Proforma
Yes
Check whether new version is required
No
Save SP/SM existing name
Updated ICDL System Model
Yes
Check whether version information has been correctly updated
No
Update SP/SM
Updated Change Log
Yes
Strike baseline for genetic fingerprint
Archive old version & save new version
Update SP/SM Change History Derivation
Review Sheet
Complete Review Sheet
Updated Review Sheet
All Relevant Elements Identified, Modelled & Documented

Figure C6-4 Level 3 : method (work ins.) level – PISCES produce natural language representation activities

This natural language representation process may essentially be regarded as a controlled and ordered approach to the collection and recording of information via the PICS and ICDL system model. Whilst this approach ensures the recording of a consistent and complete set of information about a system configuration and its associated components it also retains the flexibility of individual maintenance organisations in terms of the approach they take and tools they utilise to recover this information. This part of the ISCM process encompasses the key role within the Inverse Software Configuration Identification (ISCI) sub-process. With regard to other SCM activities, the process also addresses change control through the requirement to version the successive specific proformas; status accounting through the maintenance of a configuration change history log and, as before, an audit trail of the activities carried out is maintained. These may be summarised as:

- Identification (of **Components** and their corresponding relationships to produce varying **Configurations Abstractions**).
- Change control (versioning of the **PICS** and **ICDL System Models**)
- Status accounting (through maintenance of **Change History Logs** of the configuration evolution)
- Audit (completion of **Review Sheet**)

Tool support is provided for each of these activities within the *M*^{4} system framework. The *PISCES* tool adopts the key role in association with this process in that it defines and maintains the **Specific Proformas** that are subsequently stored as *Word* files within the *ESIB*. A simple *Text Editor* provided within the *MuMMI* framework enables the corresponding ICDL system model to be generated and updated, whilst linkage to an external *Version Control Tool* assists in the versioning of the **Genetic Fingerprints** and in maintaining the **Change History Log**. Individual **Component ID Forms** and relationship information within the **Configuration Lists** may be updated or retrieved from the *Mu*^{2}*PITS* tool and *Word* handles the **Review Sheet**.

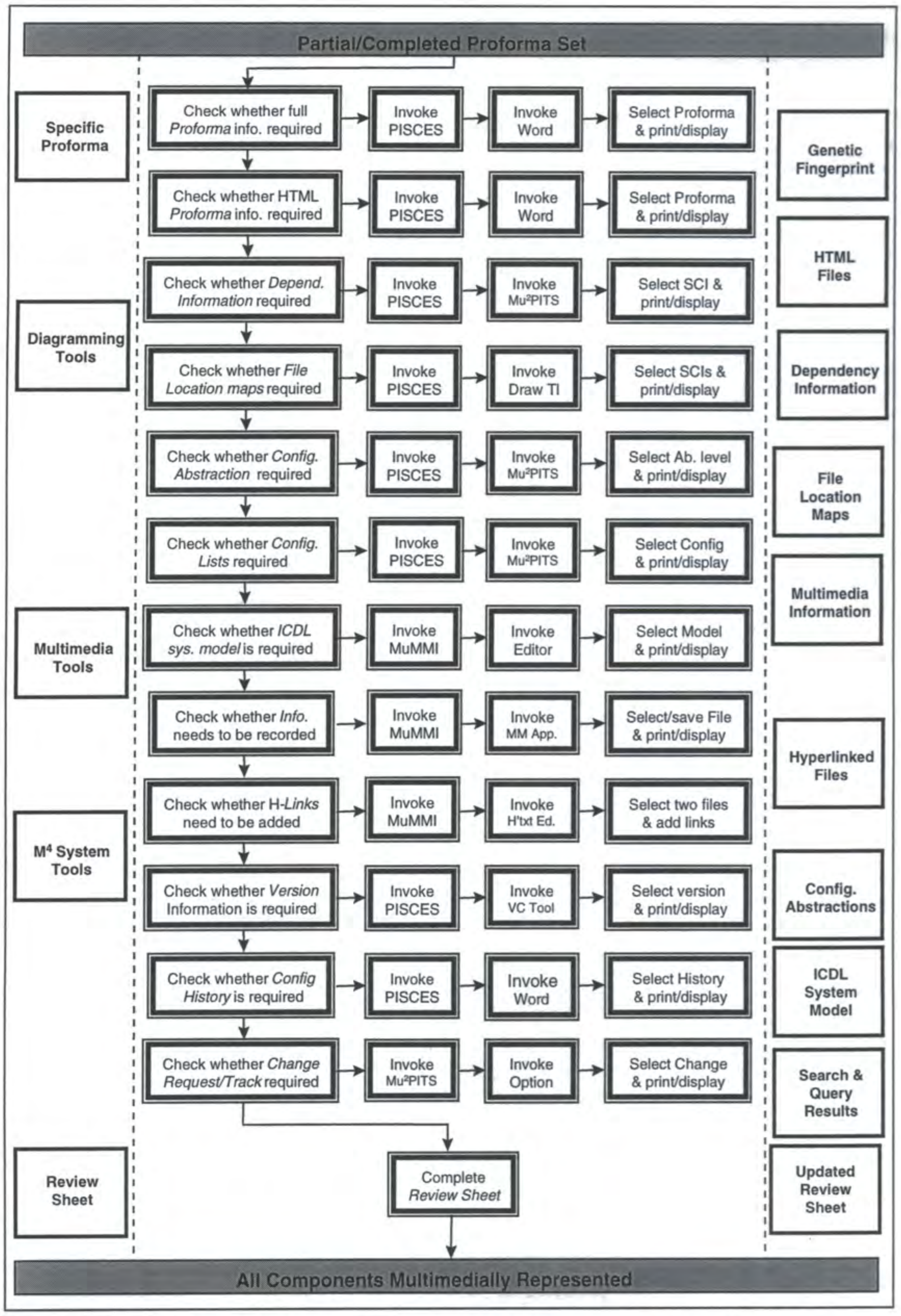
In connection with this process, the size and complexity of the applications studied were found to have an effect on the effort required in order to complete the PICS and ICDL system model. However, the extent of the effect was also determined by whether a decision was made to model the entire system or just the parts of the system affected by the change. Additionally, whilst the workload involved in physically documenting the information was directly proportional to the size and complexity of the application, the documentation actually only accounted for a relatively small part of the overall comprehension process. Indeed, the greatest consumer of effort was that attributed to the 'thinking' or analysis part of the process and hence the effect of this complexity can be generally apportioned to the corresponding analysis process. The way the M^4 system handles the increasing amount of information was also found to be appropriate for the applications studied in that the level of complexity and amount of information involved could be made visible or masked through the use of multiple windows, hypertext and hyperlinks and by considering the configurations at varying levels of complexity.

6.3.5 Process 5 – Generate system configuration output

Associated with the identification process is the generation of various reports and views pertaining to the recovered system configurations. One of the key features of the ISCM process and its implementation via the M^4 system is its innovative use of multimedia attributes such as audio, video and animation, in addition to the more traditional graphics and text. The use of multimedia files augments the incremental documentation process and enhances the representation of knowledge regained about a system during the program comprehension process. The aim of the *'generate output'* process is to guide the user through the range of possible outputs that may be produced as a result of the ISCM process, PISCES method and M^4 system implementation. For some of the activities associated with this process information may be generated automatically, for others information may be created and displayed within the M^4 system framework and, for others manual modelling must be undertaken but with the help of on-line packages for storage and update. The PISCES method activities associated with this stage of the ISCM process are summarised in Figure C6-5.

This process may be said to act as a *'table of contents'* to the types of information representation and reports that may be produced once a software system configuration has been reclaimed. The types of output are extensive ranging from the textual listing of the ICDL system model, to the graphical location maps of the identified system components to the audio and video representation of cognitive knowledge, and the animation of process information. Depending on a maintainer's requirements, some or all of these information types may be accessed and displayed on the screen or sent to a printer for a hard-copy record. In order to produce the information required one or more of the primary tools integrated within the M^4 framework are invoked and in some instances a secondary tool must be effected in order to produce or display the output. More detailed descriptions of how to produce each information type have been documented as part of the user-instructions or on-line help associated with each tool.

The provision of these core set of reports and information types ensures that a consistent and complete set of documentation about a system configuration and its associated components may be obtained. However, additional tools may be integrated within the M^4 framework to support the needs of individual maintainers. Another advantageous feature is the provision of a facility to enable the maintainer to affix hypertext links within documents and hypermedia links between information types in order to create information network appropriate to their needs.



Partial/Completed Proforma Set
Specific Proforma
Diagramming Tools
Multimedia Tools
M4 System Tools
Review Sheet
Check whether full Proforma info. required
Invoke PISCES
Invoke Word
Select Proforma & print/display
Check whether HTML Proforma info. required
Invoke PISCES
Invoke Word
Select Proforma & print/display
Check whether Depend. Information required
Invoke PISCES
Invoke Mu2PITS
Select SCI & print/display
Check whether File Location maps required
Invoke PISCES
Invoke Draw TI
Select SCIs & print/display
Check whether Config. Abstraction required
Invoke PISCES
Invoke Mu2PITS
Select Ab. level & print/display
Check whether Config. Lists required
Invoke PISCES
Invoke Mu2PITS
Select Config & print/display
Check whether ICDL sys. model is required
Invoke MuMMI
Invoke Editor
Select Model & print/display
Check whether Info. needs to be recorded
Invoke MuMMI
Invoke MM App.
Select/save File & print/display
Check whether H-Links need to be added
Invoke MuMMI
Invoke H'txt Ed.
Select two files & add links
Check whether Version Information is required
Invoke PISCES
Invoke VC Tool
Select version & print/display
Check whether Config History is required
Invoke PISCES
Invoke Word
Select History & print/display
Check whether Change Request/Track required
Invoke Mu2PITS
Invoke Option
Select Change & print/display
Complete Review Sheet
Genetic Fingerprint
HTML Files
Dependency Information
File Location Maps
Multimedia Information
Hyperlinked Files
Config. Abstractions
ICDL System Model
Search & Query Results
Updated Review Sheet
All Components Multimedially Represented

Figure C6-5 Level 3 : method (work instruction) level – PISCES generate system configuration output activities

Again this part of the ISCM process performs a key role with regard to the Identification sub-process. It also supports status accounting through the printing of the configuration change history logs and change tracking status proformas. As before an audit trail of the activities carried out is maintained. The ISCM activities associated with this process may be summarised as:

- Identification (through printing of **Component IDs, Genetic Fingerprints, Dependency Information, File Location Maps, ICDL System Models** etc.)
- Status accounting (through printing of **Change History Logs** of the configuration evolution and **Change Tracking Forms**)
- Audit (completion of **Review Sheet**)

Tool support is provided for each of these activities within the *M⁴* system framework. The *PISCES* tool adopts the key role in association with this process in that it defines and maintains the **Specific Proformas** that are subsequently printed as *Word* files or converted into HTML (HyperText Mark-up Language) for future incorporation into web pages. A simple *Text Editor* provided within the *MuMMI* framework enables the corresponding **ICDL System Model** to be generated and updated, whilst linkage to an external *Version Control Tool* assists in the versioning of the **Genetic Fingerprints** and in maintaining the **Change History Log**. **Cognitive Knowledge** can be documented using the *Text Editor*, *Word* or using any of the ***multimedia*** *application packages* that can be invoked via the *MuMMI*. Individual **Component ID Forms** and **Dependency Information** within the **Configuration Lists** may be updated or retrieved from the *Mu²PITS* tool. *Word* continues to handle the **Review Sheet**.

In connection with this process, the size and complexity of the applications studied were found not to have a significant effect on the effort required to complete the process. The amount of material related to a system was found to be more abundant for a complex heterogeneous system than for a small homogeneous application, although the M⁴ system appeared to handle this adequately. Additionally, the amount of material was also found to be a function of the length of time over which the application was being maintained resulting in more information being recorded and cognitive knowledge documented as the number of program comprehension cycles accumulated.

6.4 Assessment of ISCM process, PISCES method & M⁴ system

Section 6.3 has outlined the employment of the ISCM process, via the PISCES method and M⁴ system support, on a number of applications of varying size and complexity. However, as stated in Chapter 4, the primary aim of the ISCM process, PISCES method and M⁴ system is to assist the maintenance process by reducing the amount of time spent comprehending a system prior to a change being made. This section therefore identifies the key benefits that were found to accrue with regard to the ease with which comprehension of application systems occurred with prolonged use of the ISCM model, PISCES method and M^4 system. These may be summarised as:

- ***Increased domain understanding***: noticeably, a number of the applications studied and modelled during the research were of a similar domain type, namely that of project and configuration management. It was found that as the number of systems modelled within this domain type increased, so too did the general ease and rapidity with which an overall understanding of the software functionality could be comprehended. This was mainly due to the increased experience of the domain characteristics.

- ***Advanced start to comprehension process of systems***: as the number of systems modelled and comprehended increased, so too did the number of tailored proformas available within the PISCES framework. It was found that an application being modelled for the first time but which could be categorised within a particular domain type, could make use of a pre-existing tailored proforma, to the extent that the initial identification activities of the modelling process could be at least partially by-passed.

- ***Advanced re-start to comprehension process of systems***: particular benefits were found to accrue for systems that were incrementally modelled or maintained over a period of time. In these cases, the specific proformas associated with a particular application enabled an accelerated start to the maintenance activity to occur. This was through the documentation within the specific proforma of the attributes regarding the application components, their dependencies and the understanding gained during previous maintenance cycles.

- ***Consistent framework***: the comprehension process was also significantly eased by the provision of guidelines for the activities required during the comprehension process. In the initial 'manual' study of the RCS and SPMS applications, the modelling and comprehension process tended to be performed in an extremely ad-hoc and unstructured manner. However, this initial modelling was invaluable in determining the types of activities carried out during the comprehension process, such that they could be put into the structured framework of the ISCM process and PISCES method.

- ***Audit trail***: closely connected with the provision of guidelines for the comprehension activity was the provision of review sheets to record the comprehension activities that have taken place, when they occurred and by whom. Provision of this information was found to be useful, not so much in terms of increasing the speed at which comprehension could take place, but with regard to documenting the extent to which the comprehension activities had been carried out, a previously unrecorded factor. This influenced the degree of confidence with which a change could be made such that it avoided any potentially damaging 'ripple effects'.

- ***Increased on-line documentation***: although the PISCES method provided a solid basis on which to comprehend and model software system configurations, it was found that in order to be used effectively, even for small systems, a degree of on-line support was required. This was due to a number of factors, including:
    - ◊ the amount of information generated during the comprehension process;
    - ◊ the need to maintain the reclaimed information itself, which soon became prohibitively time consuming if recorded manually on paper;
    - ◊ the ability to store heterogeneous information types including text, graphics animations, audio and video footage;
    - ◊ the ability to link relevant information types together, and
    - ◊ the ability to search for pre-existing information on-line.

  However, once the information recovered during comprehension was recorded on-line it was found that incremental documentation of an application became progressively more rapid and effective.

The above benefits were collectively found to ease the problems of program comprehension by enabling the configurations of legacy systems to be more rapidly understood than when the comprehension process and subsequent maintenance activity took place in an ad-hoc manner. This was evidenced by the ease with which systems were being modelled, understood and updated at the end of the research in comparison to those at the beginning when the modelling process was undertaken via a manual method. Due to the application modelling being conducted as an incremental feedback process throughout the research rather than a statistical series of tests, the above benefits are qualitative rather than quantitative. However, the trials were sufficient to prove that the rate at which comprehension could proceed increased significantly as the number of applications and the extent to which these applications were modelled increased.

6.5 Summary

This chapter has described the practical realisation of the ISCM process, ISCM maintenance model, PISCES method and M^4 system in the maintenance of a number of 'real-world' applications. These applications ranged in their size and complexity from small-scale laboratory examples, through medium-sized programs to larger-scale heterogeneous systems. Depending on the size and complexity of each application, full or partial system configurations were modelled by systematically applying the activities defined in each of the process stages of the **ISCM model**. Audit sheets raised as part of the initial process were used to track the progress of the maintenance activities identified in connection with the **ISCM process** and defined within the **PISCES method**. Automated support was provided through the invocation of relevant tools or utilities implemented or integrated as part of the **M^4 system**.

Adherence to each phase of the model was shown to drive maintenance process activities from inception of a required change (*process change request*), through analysis of the affected parts of the system (*conduct analysis of legacy system*), to storage (*populate extensible system information knowledge base*) documentation (*produce natural representation of ISCM model representation*), and output (*generate system configuration output*) of the reclaimed configuration information. Although a 'paper-based' description of the model within this thesis necessitates a sequential description of each phase and implies a sequential ordering to the activities defined within each phase, in practice many of the activities were found to be conducted in parallel and iteratively. Indeed, an iterative approach to the process activities was shown to be a favourable requirement in terms of being able to incrementally document system configurations as the maintenance phase proceeds asynchronously over a period of time.

Combined usage of the ISCM model, PISCES method and PISCES M^4 system were found to be effective for reclaiming the configurations of the existing systems irrespective of their size and complexity. However, application size and complexity did impact on the degree of work associated with many of the defined activities, the extent of which was often related to the amount of information that had to be cognitively analysed or physically documented. Additionally, this high degree of human reasoning associated with analysis of a legacy system undergoing maintenance necessitated that the ISCM model, PISCES process and M^4 system, acted as guides, supplements and semi-automated support to the human-oriented maintenance process rather than a machine-driven replacement to it.

A more detailed discussion regarding the effectiveness of ISCM process, PISCES method and M^4 system on the maintenance of the trial applications is given in Chapter 7 together with the projected impact of the process and associated activities on the maintenance of large-scale industrial systems.

Chapter 7 also assesses the degree to which each of the other objectives identified at the start of the research and described in Chapter 1 have been achieved.

Chapter 7

Critical Evaluation and Discussion of the Research Objectives

Let us from point to point this story know
To make the even truth in pleasure flow

Shakespeare All's Well That Ends Well, Act V, Scene III

7.1 Introduction

The overall aim of this research has been to investigate ways of bringing existing or legacy software systems back under configuration control. To this effect the work undertaken has investigated a number of issues and proposed several part-solutions which, when combined, provide an innovative approach to the software maintenance process. To achieve the overall aim, a number of objectives were proposed and systematically investigated. These objectives progressively moved the research through a sequence of defined stages namely the:

- Conceptual definition of a process and its associated process model.
- Realisation of the process and model through development of a practical method.
- Documentation of the results from the method through a sequence of languages, proformas and review sheets.
- Semi-automation of the method through the construction of a set of CASE tools and their integration into a meta-CASE framework.
- Practical application of the process, model, method and tools through the study of a number of 'live' applications of increasing size and complexity.
- Evaluation of the work undertaken.

This chapter reviews the research carried out and assesses the degree to which each of the part-solutions set as objectives in Chapter 1 have been achieved. The exceptions to this are *Objective 9 - critical evaluation of the research undertaken* and *Objective 10 - identification of future work*. Objective 9 is met through the nature of this chapter and Objective 10 will be addressed within the final chapter of the thesis.

7.2 Discussion of Objectives

The following sub-sections summarise the work undertaken on an objective by objective basis and evaluate the strengths and weaknesses of the approaches adopted and results obtained within the scope of the thesis. Particular emphasis however is given to discussion of *Objective 1 - the definition and development of the ISCM process model* as this has been the primary aim of the research.

7.2.1 Definition & Development of the ISCM Process and Process Model *(Objective 1)*

The primary objective of the research has been the definition and development of a process and associated process model to facilitate the maintenance of legacy software systems. To this effect the ***Inverse Software Configuration Management (ISCM)*** process has been identified and defined as being:

> ***"the process of bringing existing (legacy) software systems back under software configuration control."***

Development of the ISCM process has resulted from the fusion of three key areas within the computing discipline identified as having great relevance to today's demands for cost effective, functionally correct, reliable and safe software systems. These three key areas and the reasons for focusing on them in the research are:

- ***Software maintenance***: this encompasses all activities that take place after a software product has been delivered to the customer and may account for between 40% and 85% of software expenditure over a product's lifetime [10, 49, 165, 243, 405]. The increasing size, complexity and invasion of software into other disciplines indicate that this figure is unlikely to decline. Additionally, within the maintenance process itself, up to 50% of the effort expended is in relation to the program comprehension activity which is carried out in order to understand the affected part of the program prior to making a requested change. Definition of a process that meets the needs of the program comprehension activity will realise significant benefits in terms of time, monetary and safety gains.

- ***Software configuration management (SCM)***: this is the process of controlling the evolution of complex software systems through the activities of identification, control, status accounting and audit of software system components, their configurations and any changes made to them over their lifetime. SCM is recognised as an effective control mechanism if applied throughout development and onward into maintenance [65]. However, many legacy software systems have been developed without any regard to SCM and resultantly have become '*out of control*' such that, even after a lengthy period of program comprehension, guarantees of safe maintenance changes are virtually impossible to give. Additionally, even within the traditional development-oriented SCM process, it has been acknowledged that there are still a number of fundamental issues waiting to be resolved [138].

- ***Software process models:*** these represent a conceptual sequence or cycle of stages and activities whose realisation through an associated method enables a defined process to be conducted or a specified problem to be solved. Process models for software, although developing, are still very much in their infancy, particularly with regard to the maintenance process. Additionally, the continued research and development of software process models in order to standardise the set of methods, procedures and artefacts intrinsic to the software lifecycle has been hailed as extending the 1980s decade of the methodology into the 1990s decade of the process [123].

Amalgamation of these three areas to form the ISCM process has been achieved by applying, extending and enhancing the principles and techniques of traditional SCM to the program comprehension activities that occur during maintenance of legacy software systems. The ISCM maintenance process model has then been developed to encapsulate and structure the activities intrinsic to the ISCM process.

The characteristics of the ISCM model were determined after a study of several traditional development-oriented models and a number of more recently proposed maintenance process models. As a result of this investigation, a number of deficiencies were identified with even the most recent models in connection with their superficial treatment of the comprehension of legacy systems during the maintenance process. These may be summarised as:

- ***Insufficient granularity***: it was found that the level of process granularity in many current maintenance models, although an improvement on traditional lifecycle models, is generally still at too coarse a level to ensure the controlled maintenance of legacy systems. Emphasis within the ISCM process has therefore been placed on ensuring that the process steps are defined and described at a sufficiently low level of detail. This approach enables consistency and traceability of the maintenance process across a diverse range of system and application types, whilst still allowing a degree of flexibility in the working practices and methods of different maintenance organisations.

- ***Lack of focus on identification***: it was found that many maintenance process models place their emphasis on the more downstream activities of change control, status accounting and audit rather than on the initial identification activity. The ISCM process recognises that in order to facilitate understanding of a system being maintained, there is a need to address more closely the identification of an existent system configuration and its decomposition into interrelated component parts. This realisation of the need to progressively reclaim the configuration of an out of control legacy system has led to the sub-process *Inverse Software Configuration Identification* (ISCI) being instigated. Correspondingly, the process *Inverse Software Configuration Management* (ISCM) is defined as the 'umbrella' process for the identification and comprehension process plus the subsequent control, status accounting and audit of future maintenance changes.

- ***Location mapping***: it was found that many approaches to program comprehension make the assumption that the components of the system are stored locally and occur as a coherent set of components. This is however often not the case, due to components having become dispersed across the file system such that some key components may be segregated from the main component directory and may be or may appear to be 'missing'. Alternatively, redundant or 'alien' components may have become resident within the application directory. Additionally and increasingly the distributed nature of systems development and the resultant distributed application systems such as multimedia or web based products necessitate the need for very careful file location mapping of components. Resultantly, another key feature of the ISCM model developed has been the integration of the information reclamation process with location mapping of the system components. This enables efficient access to components once changes have been approved or enables controlled reorganisation of the physical storage of components.

- ***Short-term nature of comprehension process***: it was found that one of the major inefficiencies associated with the program comprehension process was the requirement for the affected part of the system to be understood from scratch each time a change was required, even if comprehension has previously been conducted on that particular part of the system. This phenomenon can be attributed to the failure of maintainers to document the understanding they gain about a system as they proceed through the program comprehension activities. The information recovered and understanding gained is then forgotten by the time the next change is required or different maintenance personnel are assigned to the change. Since the 80:20 rule may be applied (80% of the time is spent on 20% of the code) it follows that documentation of this understanding could realise significant benefits. The ISCM method therefore also encompasses a mechanism for incremental documentation of the understanding regained about these components and their contribution to the overall system configuration. Additionally, this collection, storage and access of information pertaining to systems and changes to these systems is managed at varying levels of abstraction via a series of proformas and an enriched multimedia user interface (MUI) incorporating hyper-links.

- ***Focus on low level detailed information:*** it was found that most program comprehension tools focus their understanding on study of the program code. Whilst agreeing that the source code will always constitute the critical information pertinent to the program comprehension process, studies have shown that the efficiency of the process can be markedly improved if a mix of information types are consulted at varying levels of abstraction and different levels of formality. The ISCM process has therefore addressed this need through the identification and analysis of all associated documentation and information types pertaining to the application itself and to the environment with which it interacts from the low level source code to the high level cognitive knowledge and enterprise and business domain information.

- ***Originality of Interface***: it was found that most program comprehension tools offer a textual and graphical interface to the maintenance process, sometimes with the incorporation of hypertext. Whilst this is adequate for representing code and text, it becomes restrictive when trying to represent a very heterogeneous mix of document types. Therefore this work has extended the representation of maintenance information to incorporate video, audio and animation as well as the more traditional graphics and text. Not only does the process provide an efficient and effective mechanism for information collection, but the multimedia aspects enable a more heterogeneous representation of this information at formal, semi-formal and informal levels which result in a more intuitive understanding of the system.

The main strengths of the ISCM process defined within this thesis and its associated maintenance model thus lie in its structured approach to the program comprehension process with the primary emphasis being on identification of a system's configuration and its decomposition into the component parts and their corresponding relationships. The lower levels of granularity of the process steps enable consistency and completeness of the process, which when coupled with the retrieval and display of information at higher levels of abstraction than the source code alone provide an effective mechanism for increased understanding of the system and the implications of the proposed change. The multimedia capture and display of the progressively reclaimed information and recovered domain understanding also increases the intuitiveness of both obtaining and understanding the information. The issues surrounding the definition and development of the ISCM process and associated process model have been discussed in Chapter 4. Although the model is fairly well defined and a number of unique features are implemented, it is still at an early stage in its development and although utilised effectively during the research with several applications it may need to be refined as its usage is extended to a wider variety of application systems. The detailed realisation of the model and its effective usage are discussed within the remaining objective summaries.

7.2.2 Description, guidelines and documentation of the PISCES method *(Objective 2)*

Within Chapter 4 the ISCM process and its associated maintenance process model have been described at high levels of granularity corresponding to levels zero, one and two of the model. These levels progressively detail the main process stages of the model; the pre-and post conditions for each stage; the input resources required for each process; and the outputs generated from a particular process. However, even at abstraction level-two the process and model may still be considered essentially conceptual in nature. For the model to be physically realised, a series of detailed stages or guidelines must be defined for each aspect of the ISCM process. This provision of a detailed description or method facilitates the translation of the theoretical concepts to a real-world base. For the ISCM model this has been achieved through the definition of the Proforma Identification Scheme for Configurations of Existing Systems (PISCES) method. An explicit link between the ISCM process and PISCES method has been ensured through the continued abstraction of the ISCM process into level-three of the model which is at a sufficiently low level of granularity for the activities defined to be implemented as the PISCES method.

The PISCES method has been developed with the aim of providing guidelines and a structured approach to the core set of activities that should be undertaken if effective program comprehension is to be achieved, software system configurations reclaimed and the impacts of proposed changes on the rest of the system clarified. The PISCES method has been defined in Chapter 6 by a series of process activity charts which indicate the inputs to the process, the sequence of steps and decisions that should be considered during the program comprehension process, and the resultant outputs from the process. Although the ordering of the process charts imply a sequential organisation, the processes and to some extent the activities associated with them may occur concurrently and iteratively. Indeed, the important factor is not so much the ordering of the activities but the fact that they have all been addressed at some point during the program comprehension process. This is because through completion of each activity, a maintenance organisation can ensure that a degree of consistency and completeness has been applied to the process. The addition of the audit sheets also realise benefits in connection with the implementation of quality control procedures, on which increasing emphasis is being placed by many organisations.

The PISCES method could have been much more explicit in its demands, however user-centred task analysis revealed that too prescriptive an approach can act as a negative factor rather than a positive driver for maintenance productivity gains. This can be attributed to factors such as resentment at the removal of task creativity; the failure to take into account valuable intrinsic maintainer expertise; and neglect of evolved and well-tested maintenance practices within an organisation. The PISCES method has therefore been developed as a trade-off between prescription and flexibility in that the method defines the controlled and accountable set of activities that should be carried out, but their sequence of execution and the way by which they are achieved is maintainer-driven.

The PISCES method has also tried to address the balance between providing practical support for the technical activities of the maintainer and the reporting activities required by the manager. In this respect technical activities are supported by procedures and tools for activities such as data extraction, incremental documentation, file location mapping and definition and recording of configuration abstractions. Management aspects include the ability to provide change log histories, to highlight candidate areas of code for cost-effective re-engineering, and to record an audit trail of maintenance activities. As mentioned above, this accountability and adherence to a well defined process are likely to become increasingly important as developers and maintainers become held more accountable for the work they do and any adverse consequences of this work.

Within the limits of method flexibility, another level of detail could have been added in connection with the (semi-automated) support provided through the M^4 system. However a decision was made to retain the detailed procedural instructions associated with the use of each tool within the individual user guides. This decision was taken to accommodate changes in the toolset composition, but also because the individual tools are still undergoing development and hence update and enhancement of the tools themselves is likely to occur at a faster rate of change than the higher level activities of the method or process.

Practical application of the PISCES method was achieved through the modelling of a number of applications ranging from small laboratory programs to relatively large UNIX applications and the M^4 system itself. The results of these trials are discussed more fully in section 7.2.8, however initial findings suggest that it is relatively easy to apply the activities defined within the PISCES method to a range of applications. As the applications were modelled, feedback regarding the effectiveness of each process was made possible and the activity descriptions and their suggested sequencing were adjusted accordingly. Whilst the method proved to be an effective guide to the modelling and comprehension of all applications studied, it is recognised that this has been based on a very limited sample size. It is therefore likely that the method will evolve as it is used in connection with a greater number of applications of different types and from a wider range of domains. It is feasible that a number of variants of the method will be necessary in order to accommodate all software systems. This is because some types of system may require greater emphasis on particular aspects of the model, or a more rigidly defined approach than others. For example, safety critical real-time systems may need to be modelled differently to business batch-oriented transaction systems. These model variants could be defined and maintained in a similar fashion to the generic, tailored and specific set of proformas.

7.2.3 Modelling Software System Architectures through Abstraction *(Objective 3)*

In order that legacy software system configurations can be identified there is a need to progressively model the components of the application system and the relationships that define their composition into viable configurations. The ISCM process has based its approach around the definition of a number of component-type groups, the components of which can subsequently be combined to create or identify the software system configurations at varying levels of abstraction. A review of what constitutes software system architectures was conducted to enable the ISCM core component groups to be identified. The resultant inclusion of a more enriched set of component types than any of the system models reviewed contributes to the uniqueness of the ISCM approach. The originality of the approach also extends to the modelling of software configurations as a series of maintainer-definable configurations abstractions. These abstractions are based on the interaction of 'baseline' application group components with components from other groups, thereby placing emphasis on the modelling of an application within the context of its environment. Modelling in relation to the system environment was considered to be an important issue given that it in relation to today's software systems it has become very difficult to consider the application in autonomous isolation.

Linked with the above, one of the key premises on which the ISCM approach is based is the ability to model systems at varying levels of abstraction. Software systems and the associated modelling of their configurations have become progressively more complex over the past decade. This complexity may be largely attributed to the resultant effects of advances in hardware, software and communications technology and include:

- ***Implementation shift***: technology advances have enabled software to take on traditionally hardware-oriented roles through task implementation using complex algorithmic bases .

- ***Implementation evolution***: technology advances have resulted in far more advanced human computer interfaces, parallel processing algorithms and heterogeneous multimedia components being incorporated into software systems. Additionally advances in communication technology have resulted in the number of distributed applications across multiple and often geographically dispersed platforms. In particular the meteoric rise of world wide web based activities has introduced issues surrounding the maintenance of products as a series of conceptual and rapidly changing links, rather than a set of physical objects over which the owner of a system has control.

- ***Computer abundance***: widespread acceptance of computer technology has resulted in computer applications impacting on every area of society, business and industry. This has had two major effects. Firstly, in order to sustain the rate of requirement for cost-effective, safe and reliable computer systems, emphasis is being placed on the integration of reusable components within the core application. Secondly as the number of software products in the market place increases so does the computer 'food chain' in terms of the requirement of one vendor's product by another for a completely functional system. For example, a given processing or control application may require an underlying third-party database in which to store its data and a third-party front-end interface through which to access its data and control functions. Hence it is no longer the norm for components requiring modelling to come from the application system alone.

Abstraction is a technique for dealing with complexity by masking irrelevant details, either through defining the level of granularity of the information displayed or by defining different views of the partial or complete system. Both of these approaches have been shown to be effective for the modelling of large-scale software systems and both have been addressed within the research. The modelling of systems through progressive levels of granularity will be discussed in Section 7.2.6. The remainder of this section discusses the modelling carried out through defined views of system configurations.

As mentioned above, modelling within the ISCM process is based on identification of a comprehensive number of components groups and their interaction with the members of the application component. Although some work has been done in this area, it has tended to be limited to modelling program family versions, build-tool characteristics and occasionally the documentation associated with a software system. Review of the literature did not reveal any work that examined the same level of modelling detail as the ISCM process. Nor does the other work reviewed use the concept of describing software by a series of definable and re-definable configuration abstractions. In the practical trials of the method and approach described in Chapter 6, targeted modelling of the relevant aspects of the system was found to be an effective aide to the speed with which application systems could be understood during the program comprehension process.

Definition of the default component group membership and configuration abstraction composition is made within the PICS and ICDL. The ability of the maintainer to redefine the default memberships of the configuration groups was also considered important within the context of the process. The reasons for this were two-fold. Firstly definition of group membership was found to be somewhat subjective in nature since a number of components types could have been placed in more than one group. Secondly, the nature of the components and the roles they play depends to some extent on the nature of the application being maintained. For example, the relationship of an application component to its environmental counterparts may be very different in an embedded system than in a data processing application. The configuration abstractions can also be redefined within the PICS and ICDL system model templates to enable modelling of the most appropriate views of the system.

The modelling has also initially been kept very simple, with the definition of the abstractions being limited to unions of entire component groups. In reality, modelling is a more complex process than this implies in that only sub-sets of the component groups will produce viable system combinations. The extent to which these should be defined and modelled is a candidate for future refinement of the model. Similarly, within the scope of the thesis, the modelling of the relationships has been limited to those existing between the application components (intra-application modelling) and to those between the application component group and the other component groups (inter-application modelling). There is however, another level of complexity associated with the modelling process in that interrelationships may also exist between the various non-application component groups. However, the complexity of modelling at this level of detail was considered to be outside of the scope of the thesis. It was also assumed that the component relationships remain fairly stable and traceable once modelled and hence issues such as web-based product maintenance have not been addressed. In such applications it may be impossible to trace components once a referential link has been broken due to movement of the component by another owner. However, this is a very relevant area of interest and one that will be addressed in the future.

7.2.4 Definition and Structure of Underpinning Information Base *(Objective 4)*

The information regained about the software system configurations and their associated component parts must be stored in a format suitable for retrieval, reference or update if the effort expended during maintenance is to be effectively reused. This is an essential requirement if one of the major inefficiencies of the maintenance process is to be addressed, that is, the time spent performing program comprehension activities, even if the affected part of the system has been maintained and comprehended before. This inefficiency can be attributed to the knowledge gained during a particular maintenance cycle remaining 'personal' to the maintainer rather than being documented as part of a 'maintenance oracle' [268] for future reference. To address these issues an Extensible System Information Base (ESIB) has been defined. The roles of the ESIB are thus to store the information reclaimed about a software system during maintenance and to store the rules and guidelines associated with the collection of this information. More debatably, another role could be the storage of the actual components of the system being maintained. The label 'extensible' has been tagged to the nomenclature of the knowledge base in order to emphasise the incremental and progressive population of the ESIB with information and new information types as they are recovered during the maintenance process.

Two approaches to the construction of the knowledge base were considered. Firstly a single central repository, with explicitly defined data structure formats could have been developed into which every aspect of information relating to the comprehension of a system must be physically stored. The second approach considered, still retained the concept of a central repository but, was essentially logical in nature with physical implementation occurring through a series of repositories whose data structure and maintenance are governed separately by the individual tools. With respect to making this choice a study was conducted regarding a number of different approaches to the development of knowledge bases within the context of achieving the goals of the ISCM process.

As a result of the investigation a physically disparate but logically coherent approach was eventually adopted. There were two main factors that influenced this decision. Firstly, the disparate approach provided greater flexibility within the context of the ISCM process. Secondly, the development of an object base that has the ability to store the highly structured and variable datatypes associated with the ISCM process was too great a task for what was essentially a 'support activity' within the scope of the thesis. Development of a more sophisticated storage mechanism can be investigated at a later date.

The flexibility afforded by allocating data storage responsibility to the individual tools removed any restrictions on the types of tools that could be incorporated into the M^4 meta-CASE framework. However, there were also a couple of negative issues associated with this approach, particularly in terms of a degree of redundancy being associated with the storage of some information and resulting in concurrent maintenance of multiple datasets. The most obvious example of this is the requirement to maintain the PICS and the ICDL system models in parallel for a particular system. A short term solution is being developed that will enable the importation of the ICDL data directly into the PICS or vice versa. However this is not yet at a stage suitable for incorporation into the M^4 system.

The ESIKB thus has been implemented as a combination of:

- ***Flat files:*** for the ICDL system model storage and representation.
- ***Relational database tables:*** for storage of details regarding the individual components, their composition into configurations, change proposals and change tracking forms.
- ***Individual files within the underlying operating system:*** for storage of the proformas and multimedia file types.
- ***Hyperlinks:*** to link the proformas to other information types.

Currently, the rules associated with the reclamation of knowledge pertaining to a particular system or category of system are represented within the proformas themselves. Eventually, a separate rule base will be established, having its own set of heuristics and incorporating enabling technologies such as neural networks, fuzzy logic and intelligent agents. Incorporation of such features will enable more intelligent and automated knowledge elicitation and may eventually lead to a situation of 'auto-maintenance'.

7.2.5 Definition of Inverse Configuration Description Language *(Objective 5)*

Widespread use of systems such as those based on the language Mesa [231] and its derivatives such as C/Mesa are said to be hindered by their language and machine dependence [369]. Hence one of the primary aims of the ISCM research has been to develop a language which is independent with regard to application system type, application language and application platform. The Inverse Configuration Description Language (ICDL) has thus been developed such that it provides a generic approach to the modelling of software systems.

A review was undertaken of a number of approaches and languages used to model software system configurations. These were found to range from very simple text-based syntactic descriptions, to more structured abstract-datatype/object-oriented definitions, to strictly defined mathematical denotations of the relationships existing between components parts of a system. The languages used for defining software system composition have in general been based on the concepts of programming-in-the-large (PITL), whereby the interface of the components and the resources required by and provided to these components are modelled in such a way that the implementation details, characteristics of programming-in-the-small (PITS) approaches, are hidden. Whilst this approach has proved to be very apt in the past for modelling software system configurations, there is an argument based on the increasing interaction of application components with their environmental counterparts, for an additional level of modelling if system configurations are to be adequately described and controlled. This concept, which translates to modelling an application system within the context of its development and functional environments, has been developed within the thesis and is termed programming-in-the-environment (PITE). The additional attributes and features required for achieving this level of modelling have been incorporated into the ICDL and hence form the primary basis by which the ICDL may be distinguished from most other languages for describing system configurations.

The ICDL defined is based on seven levels of information definition (*domain, environmental, location, total, abstraction, configuration* and *component*) which together describe the syntactic composition and semantic meaning of the components, their combination into configurations, their functionality and their interaction with the environment. Whilst some PITL languages, such as PCL [360] and REBOOT [396], offer elements of the above, ICDL provides a more comprehensive attribute set for domain and environmental description than current languages. This emphasis has been made possible through modelling configurations as a series of different abstractions and by developing the ICDL as a PITE language rather then an aggregate of PITS, PITL and PITE concepts and constructs. It is however intended that PITL and PITS languages may be used in combination with the ICDL as required and as such present a balance between prescription and flexibility of choice.

Another key way in which the ICDL differs from existing languages is in relation to the role it plays in the software configuration management (SCM) of a system. Within traditional SCM the architectural description of the system acts as a blueprint to drive the composition process. Indeed, after each maintenance change or set of changes there will be the requirement to rebuild the software system in order to create a viable configuration capable of being executed. The ICDL is not intended to carry out this role of automatically reconstructing and verifying system builds. Rather, the resultant ICDL architectural description becomes the product of the ISCM process, whose activities are directed towards incrementally recovering an unambiguous system description from a previously corrupted or disarrayed one. That is, the inverse approach involves progressive or selective decomposition of a system configuration in order to understand its composition and working, rather than being a mechanism for system composition from its constituent parts.

The aim of the ISCM process and the ICDL is thus to aid the comprehension process in order to increase the likelihood of making a safe change. However, for maximum effectiveness, once the ISCM process has assisted in the prediction that a safe change can be made, a more traditional PITL language could be invoked to initiate the system build in the normal manner. The ICDL as such acts as a precursor to the traditional system build process. Again this approach promotes flexibility, this time in relation to the build approach adopted by individual maintenance organisations or that required for different application types or platforms.

The chosen syntactic structure for the ICDL is that of a very simple text-based description. Whilst such an approach lacks the ability to represent and read complex structures as complete records or objects, it has again been devised with flexibility in mind. An important requirement for the ICDL was that it could be easily machine-read or parsed in order that the data contained within could be manipulated, displayed or reported. The format of:

```
/section-header/ information-id : results
```

with optional iteration at any point enables the file to be readily parsed by small bespoke or commercial utilities and the requested data to be extracted and manipulated by different tools, for example, the production of file location maps or configuration lists.

Emphasis on describing software configurations within the context of PITE may thus be considered an innovative approach to the description of software systems. However it is recognised that the ICDL will need to continue to evolve as more systems and system types are modelled.

7.2.6 Definition and Creation of the Proformas for Natural Language Representation of the ISCM Model *(Objective 6)*

The core element of the ISCM process is the definition and use of the Proforma Increasing Complexity Series (PICS). The series was developed as a result of an initial manual study which investigated, for a number of applications, the types of information that could be usefully recovered pertaining to maintenance of a software system. The role of the PICS is threefold:

- ***Input/output mechanism***: whilst the ICDL provides a convenient machine-readable format, the PICS present a more human-oriented mechanism for information collection, collation and reporting. The PICS also enable the input and output processes to be conducted in a manner which is consistent and comprehensive yet flexible enough to meet the demands of different maintainers or maintenance practices.

- ***System blueprint***: the proformas act as the system description document and hence as the definitive guide to the composition of a current system configuration, the identification of the relationships of the configuration to its operating environment, and a record of reclaimed domain and semantic information about the system.

- ***System life-cycle derivation***: if baselines of the system composition as represented by the specific proformas are struck at appropriate intervals, when viewed as a time-ordered sequence, they provide a record of the evolution of a system or the maintenance pattern of either part of all of a system.

In essence, all information identified as being relevant for the effective maintenance of a system should be represented and recorded within the proforma series. However, due to the large and complex nature of software systems, information elicitation and reclamation is generally a progressive process. To accommodate this, a series of three proforma abstractions (*generic*, *tailored* and *specific*) have been defined, each of a similar structure and format but which have been designed to become syntactically and semantically more attuned to a particular system as the series is progressed. That is, via the PICS the recorded information is transformed from being of an application and environmentally independent nature (*represented by a single generic proforma*), to a stage of application independence but environmental dependence (*represented by one of several tailored proformas*), to ultimately a state of complete application and environmental dependence (*represented by one of many specific proformas*). This latter stage may be considered to represent the *genetic fingerprint* of a system at a particular point in time.

Although the use of proformas or templates to describe software system configurations is not an entirely new concept, the PICS has been designed to extend the facilities offered by such techniques. This enables the PICS to effectively address the configuration management requirements of today's complex computer systems and particularly those with respect to applying the ISCM process to such systems:

- ***Heterogeneity***: the PICS, like the ICDL, has been designed to incorporate a broader spectrum of information than has been addressed by many previous configuration management ventures. Collection of this information is organised into six major sections which correspond to those fields defined within the ICDL, namely: *header, environment* & *domain, resource & location, configuration abstraction, component identification* and *components*. Completion of these respective sections enables a full or partial system configuration to be progressively recovered through a series of activities related to the identification, description, definition, specification, extraction and deduction of information.

- ***Structure***: a number of key distinctions can be made between developing/maintaining a knowledge base of information pertaining to software system configurations and a database relating to more traditional transaction-oriented records. These distinctions stem from differences in the numbers of records maintained; the complexity of the records; and the frequency of access to these records. Traditional databases are generally designed to deal with extremely large numbers of relatively small and simple records which may be frequently accessed for purposes of information retrieval or small updates. Conversely, a knowledge base such as that underpinning the PICS has a 'relatively' small number of records but which are highly complex in their structure and which tend to be accessed less frequently but for longer periods of time and for more lengthy update. The heterogeneous nature of the multimedia data also requires additional thought for effective processing.

  The templates within the PICS series, which in their entirety form highly complex structures, have thus been designed and implemented to take account of such factors. The primary way in which this has been achieved is by storing the configuration data within the PICS as a combination of physical entities and logical links. This enables any short textual descriptive data to be stored as part of the PICS, whilst allowing complex items of data such as multimedia files to be stored elsewhere on the system or distributed across a network. These complex items of data (whether *records, documents* or *multimedia files*) are accessible through activation of the hyperlinks embedded within the PICS. For example, links can be established from the PICS to *configuration id records* stored within an Access database. In this way traceability can be established from the proformas to all components identified as having a relationship to the configuration.

The workload associated with the input of the large amount of configuration information generated for a system and its subsequent display has been addressed through the implementation of the PICS as a forms-based interface incorporating elements such as *radio buttons*, *check boxes*, *pull-down lists* and the previously mentioned *hyperlinks*. These features within the PICS have enabled the high information load of large systems to be collated, documented, managed and structured whilst at the same time retaining flexibility and reducing complexity. Additionally, the use of hyperlinks has kept redundancy and maintenance of information to a minimum since links may be made from several proformas to a single stored copy of physical data.

- ***Incremental***: one of the key features of the PICS is their ability to facilitate incremental documentation of a full or partial system configuration each time the maintenance process is undertaken. The PICS have therefore been designed to be progressively populated as maintenance proceeds and for baseline or '*genetic fingerprints*' of the system to be taken at periodic intervals in order that a configuration history of the system may be established. The PICS themselves also exhibit the properties of flexibility and extensibility in that certain properties such as the composition of the configuration abstractions or definition of certain fields may be altered or created as required for particular systems or analysis purposes. In this context the incremental nature of the PICS means that data for the newly defined fields may be added at a later comprehension state.

- ***Reuse***: one of the key aims of the ISCM process is to model as wide a range of application systems and system types as possible. The PICS have therefore been structured such that one set of proformas can be used to represent the configurations of all possible system types. Through the definition of a number of generic rules and selection possibilities, which become progressively narrower as the proformas become more application specific, it is possible to reuse partially or fully completed proformas at the generic or tailored levels of abstraction. Indeed, it is not until the specific level of abstraction that there is a direct one to one mapping between the proforma and the application. This approach is very efficient in terms of documenting system configuration since a high degree of reuse is afforded in terms of the structure of information, the selection and recording of data and the information content itself. Indeed, by maintaining a library of partially completed proformas an accelerated start can be made to the program comprehension process itself. The use of referential links within the PICS also increases the level of reuse since items of documentation or notes relevant to more than one system at whatever level of abstraction can be attached to more than one proforma.

- ***Reverse Engineering***: the PICS also form a solid foundation for subsequent maintenance activities and/or configuration control of the system. One of the issues pertaining to legacy systems is that of reverse or reengineering. The PICS and ISCM process in general may be considered to assist the reverse engineering process in two ways. Firstly, the reclamation and understanding of information pertaining to a software configuration is an essential first stage in the reverse engineering process and hence the PICS form an integral part of the reverse engineering process itself. Secondly, one of the key factors often associated with reverse engineering is that of the very high costs involved. Hence, there is often the need to identify systems or parts of systems as being prime candidates to undergo reverse engineering. The information recorded via the PICS and ISCM process can help determine which systems would benefit most from reverse engineering. In particular scrutiny or comparison of the time-ordered genetic fingerprints of a system reveals the maintenance characteristics of a system and may highlight troublesome areas of the code.

- ***Consistency and documentation:*** one of the problems often associated with maintenance and addressed by the ISCM process is the lack of guidelines or defined process steps to enable a consistent approach to be taken to maintenance and the program comprehension process in particular. Definition of the PICS for use in conjunction with the ISCM method has enabled a degree of consistency and completeness to be conveyed to the maintenance process. Records of the configuration information recovered are evident from the PICS and visibility of the maintenance process available through the audit sheets, both of which are implemented on-line and may be stored electronically or as printed copies.

The PICS thus play a vital role during the ISCM process from a human-oriented perspective in facilitating the reclamation of software system configurations. The PICS have been shown to effectively guide and control the modelling of a number of application systems and hence have met their primary objective. Additionally, they have been shown to be able to accommodate a wide range of application types and hence meet the requirement for an independent, generic and flexible framework. However, the PICS still require further refinement and possible extension in order to improve aesthetics, maximise effectiveness and encompass more application domains, system environments and programming languages etc. There is also the need to increase the comprehensiveness of the knowledge base such that information pertaining to a system can be automatically recovered and inserted into the proformas. The conversion of the proformas into html (hypertext mark-up language) has also begun and their usage for maintenance across the web being investigated. The main criticism of the work to date is the requirement to maintain in parallel the machine-oriented ICDL and the human-oriented PICS. Work is however underway to resolve this and once completed will be implemented within the M^4 system.

7.2.7 Implementation of the PISCES M^4 System *(Objective 7)*

The PISCES method may be effectively applied as a sequence of stages to facilitate bringing legacy software systems back under configuration control. However, although adoption of a method alone can realise significant benefits, there is generally a need to provide computerised support if it is to be usefully applied to large real-world systems. This is particularly true in relation to the software maintenance discipline, where many of the problems to date have typically arisen from deficiencies in recording and being able to easily access any knowledge regained about a system during the program comprehension activity. It is also imperative that any method adopted is viewed by maintainers as assisting rather than hindering them in their work. Again, an element of semi-automation and on-line support is regarded as essential in ensuring that this requirement is met. With these factors in mind a series of prototypes have been designed and developed during the course of the research culminating in the creation of the *PISCES M^4 (MultiMedia Maintenance Manager) system.* In order to promote maximum process flexibility and product extensibility the M^4 system has been created as a meta-CASE framework into which a variety of bespoke or host-resident tools may be plugged and activated though a universal front-end. This 'open' approach is becoming increasingly popular in the CASE marketplace [114, 181]. The core toolset of the M^4 system currently comprises:

- ***Extensible System Information Base (ESIB)***: for storage, querying and recovery of rules, component details and domain knowledge.

- ***Proforma Identification System for Configurations of Existing Systems (PISCES)***: for collection, collation and dissemination of information via the provision of on-line proformas and guidelines.

- ***Multimedia Application Documentation Domain (MADD)***: for access to, management and control of configuration documentation.

- *Multimedia Multi-Platform Identification and Tracking System (Mu^{2}PITS)*: for recording information about individual software items, tracking their change status and their composition into viable configurations.

- *MultiMedia Maintenance Interface (MuMMI)*: as an overall controlling interface, invocation of the core toolset plus any host resident tools responsible for activities such as version management, configuration building, information extraction or incremental documentation using multimedia attributes.

Tool support for the maintenance process, although growing in stature, is still small compared with the number of tools available to assist the development process. The M^4 system has thus been designed to add to the set of tools available for maintenance and in doing so also offers several advantages over many maintenance tools currently on the market. These may be summarised as:

- ***Role***: the primary aim of the M^4 system is to support the ISCM process and PISCES method. As the process and method have been defined within this thesis as the subject of the research, the M^4 system is unique in its ability to support the process and method in their entirety.

- ***Flexible***: adoption of a meta-CASE approach to the development of the M^4 system has enabled it to be very flexible in terms of its functionality, and readily extensible with regard to the range of tools that it can incorporate. This enables the system to be efficiently adapted to a particular operating platform since any 'redundant' tools may be unhooked and replaced with more appropriate tools for the particular environment. In contrast many current maintenance tools are far more prescriptive in their functionality and are difficult to adapt or extend once they have been bought and installed.

- ***Generic***: the approach taken by the PISCES method and implemented in the M^4 system also means that as well as being adaptable for different operational platforms, the system may be tailored to suit different application domains and programming languages through the on-line implementation of the generic, tailored and specific proformas. In this respect the M^4 system may be considered to be more generic than many current maintenance tools, although it must be stated that this is a part-result of the generic nature of the method itself.

- ***Cost effective***: many current maintenance tools are costly to buy and implement. In addition to actual tool expenditure, there may be the requirement to reorganise the computer system on which the tool is to be run, or to change current working practices, if the tool is to be used to its best advantage. The M^4 system however can be integrated with tools already residing on the host system thereby reducing cost, and minimising the disruption associated with training maintainers to use new tools. Additionally, as the PISCES method provides guidance rather than prescription for the maintenance process, the different working practices of maintainers can be accommodated within the framework of the system whilst still providing the required consistency and control. This ability to integrate with the users environment with minimum disturbance is considered essential to the acceptance of the M^4 system.

- ***Multimedia***: although the use of text and graphics and their linkage through hypertext are now commonly used within tools of all types, the ability to exploit the capabilities of multimedia (and when linked together, hypermedia) within software maintenance tools does not appear to have been comprehensively investigated to date. However, the popularity of audio and video as an information source is increasing and the concept of what constitutes a document changing accordingly [204]. In response to this, the M^4 system uniquely makes use of a hypermedia approach to enrich the maintenance and continued development of software systems. It also provides the MuMMI interface as a means of managing the recording, activation and dissemination of multimedia material pertaining to software system configurations. Indeed development and evaluation of the M^4 prototypes has demonstrated the feasibility of using multimedia technology to gain real benefits in the area of program comprehension. Initial interest in the product has also confirmed that with the correct approach, the use of a multimedia system for development and maintenance could become widely accepted by the computing industry, particularly with the widespread integration of multimedia technology into almost every aspect of our working and social lives.

- ***Domain Knowledge***: The M^4 system pays particular attention to being able to capture domain knowledge about a system undergoing maintenance. This is facilitated by exploitation of the multimedia capabilities described above. Indeed, the importance of having a tool that can incorporate domain expertise is emphasised by studies which found that it is 'better to have specific knowledge about the problem domain and a weaker programming mechanism than a more powerful mechanism and general knowledge' [290].

- ***Communicative***: the key findings of another study [189] were that maintainers expressed the need for better communication between the development and maintenance teams. This included the need for the initial transfer of knowledge between the two teams to be as complete as possible, as well as requiring the long term conservation, dissemination and refinement of expertise from one maintainer to another. The long term transfer facility is provided by using the M^4 system during maintenance whilst initial transfer of knowledge is also facilitated if the M^4 system is used from the outset of the development process.

- ***Transferability***: a by-product of the M^4 system development is the transferability of the underlying framework into other realms of information management. Although the M^4 system has centred around providing an environment for the development and maintenance of software systems it has become evident that there is enormous potential for expansion of the ideas into many other application areas requiring production and control of a mixed media type. The feasibility of transferring the ideas and associated technology of the M^4 system into other areas of information management is already being commercially exploited in some domains and investigated further in others.

Although a successful working prototype exhibiting the above characteristics has evolved during the course of the research, there are still a number of ways in which the M^4 system can be improved both in relation to the degree of functionality offered by the M^4 system and in relation to the quality of development of the prototype system. These may be briefly summarised as:

- ***Functionality***: a series of progressively more sophisticated tools have been integrated into the M^4 system during its evolution, however these still require a high degree of human involvement in their invocation and interpretation of the resultant data. Improvements may thus be made with regard to the degree of automation provided by the system in terms of information extraction and inference. In connection with this is the need to further build up the information within the ESIB in order that enough data exists for maintenance patterns to be established both about generic system types and specific applications. This will also involve building up a knowledge base of partially complete proformas together with an effective indexing system to enable their effective reuse.

The degree of granularity of the hypermedia linkages also needs to be refined. Currently, these linkage act at a file to file level, however for maximum effectiveness the linkages between different media types should be at a point to point level. Work is well underway to enable this.

There is also the need to add/further develop the tools that are integrated into the toolsets in order to aid the extraction of the system information and its subsequent representation.

- ***Quality***: there are a number of observations to be made from an implementation point of view which also need to be addressed. Although emphasis has been placed on efficiency of the M^4 system such that there is little delay in relation to the access, retrieval and display of the multimedia attributes, other areas of performance and quality need to be addressed. For example, little work has been done to date on the provision of verification mechanisms for the data being entered into the system. However, this area is being addressed in a commercial variant of the system.

  Additionally, the individual components of the core toolset have been developed on a rather piecemeal basis using a variety of languages ranging from C to Visual Basic. Whilst this exhibits the ease with which different tools may be integrated together within the M^4 framework, a degree of uniformity in development strategy for the core toolset is seen as desirable and an investigation into developing the entire system using Delphi or Object Builder is underway. This would also enable a more secure 'bonding' between the core tools to be made.

  There is a need to address the issues of redundancy arising with regard to some of the data stored within the system particularly with regard to the ICDL and the PICS. Finally, there is also the need to test the M^4 system more extensively on a much larger number of documents of each media types, particularly in connection with large applications.

Although, the above highlight a number of deficiencies with the M^4 system, it must be remembered that it was developed as a prototype to assist implementation of the ISCM process and method. Many of the highlighted issues are already being addressed in the commercially developed variant of the system. The effectiveness of the M^4 system for assisting the program comprehension process when used in conjunction with the ISCM process and PISCES method is discussed in the next section.

7.2.8 Practical Application of the Process, Model, Method and Tool *(Objective 8)*

In order to assess the effectiveness of the ISCM Process, PISCES method and M^4 System for modelling real-world systems a number of test-bed applications were used. These ranged from very simple laboratory sized programs to more complex real-world systems and culminated in determining whether the approach would be suitable for maintaining the M^4 system itself. The degree to which Objective 8 has been met may be discussed from joint viewpoints. Firstly, the approach adopted for evaluation of the ISCM process and its corresponding model, method and tool, may itself be evaluated. Secondly and more importantly, the findings of the ISCM evaluation in terms of its effectiveness in assisting the maintenance of real-world systems may be deliberated.

As mentioned in Chapter 6, evaluation of the ISCM process and its associated model, method and tool has been cumulative over the duration of the research rather than a single post-development activity. As such it has been interwoven with the development process and based on a number of techniques such as manual modelling, prototyping, iterative and evolutionary development, walkthroughs, metrics collection, and user analysis and evaluation. This approach was considered to be appropriate for several related reasons:

- ***Solid foundation***: manual modelling of the target applications assisted in formulating the requirements for the overall ISCM process; identifying the components for modelling systems at a programming-in-the-environment (PITE) level; determining the activities for the ISCM model and PISCES method; and acting as a comparable test-bed for the semi-automation of the process through the M^4 system.

- ***User-centred***: there is always a danger when developing a model, method or tool that the end-product may not address sufficiently well the task-needs of the user. Incorporation of front-end user requirements capture and on-going user evaluation of the method and tools has enabled the products to be tailored for maintenance specific features and activities.

- ***Early and continuous deliverables***: the prototyping approach enabled early deliverables to be available for the purposes of requirements elicitation, user evaluation, and feedback into better or more focused development and commercial exploitation of some of the underlying features of the system. On an individual tool basis, the tools were occasionally rapidly prototyped, discarded and then rebuilt, but the majority were developed on an evolutionary basis. In terms of the overall system an incremental development approach was adopted whereby the up-front analysis of the system was followed by staged implementation of the individual tools.

- ***Feedback & control:*** The decision taken to conduct the evaluation as an iterative activity throughout the research also proved to be successful in terms of the resultant feedback. This enabled adjustments to be made to the process, model and system, resulting in a more maintainer-oriented product and reducing the risk of having to make significant changes late in the development process.

- ***Analogous development to product strategy***: part of the ethos of the ISCM process and associated tools is the flexible, incremental and iterative nature of the process itself and the accompanying toolset, in order that they may both be as generic as possible. By developing, in this iterative manner, the ability to adapt the model for differing systems and domains and to add tools incrementally to the toolset, the approach was found to complement the intended strategy.

- ***Transferability & scalability***: within the timescale of the project it was only possible to model a relatively small number of applications. However, these differed in their size, complexity and composition and were capable of showing the effectiveness of the results of the research, particularly with regard to the differences between modelling very small applications and those of a much larger and more complex nature.

The ISCM model described in Chapter 4, may be regarded as a high level representation of the ISCM process. This level of abstraction is suitable for providing an overall impression of the activities of the process to customers or project managers, however for guided and controlled use of the model by maintainers a lower level of detail is required. This is provided through the combination of the PISCES (Proforma Identification Scheme for Configurations of Existing Systems) method and the M^4 (Multimedia Maintenance Manager) meta-CASE system. Definition and application of the activities encompassed within each major process of the ISCM model has enabled the applications studied to be reconfigured, comprehended and brought back to full or at least partial configuration control. The advantages offered by the PISCES method and M^4 tool may be summarised as:

- ***Genericity***: the process and model proved to be suitable for modelling applications of different sizes and complexities. Additionally, the ability (within limits) to extend or mask details on the PICS (Proforma Increasing Complexity Series) as maintenance proceeds over the lifetime of a product or in response to new applications means that the approach has potential for universal application across a range of systems of disparate types.

- ***Flexible & encompassing***: the method is intended for incorporation into an existing maintenance process framework and hence is very flexible in terms of the maintenance practices and procedures that it can encompass.

- ***Presentation of varying abstractions of information***: the approach adopted also made it possible to present different levels of information or different information types to 'client's (customers, managers, maintainers) in response to their differing requirements.

- ***Incremental***: the method and system provided the means for incrementally documenting and storing within the ESIB (Extensible System Information Base) the knowledge regained about a system during the program comprehension process such that it is made available for use by future maintainers. This has the effect of transforming the comprehension part of the maintenance process from an expensive 'overhead' to that of a 'value-added' activity.

- ***Concurrent documentation and maintenance***: the ability to record maintenance activities using multimedia attributes such as sound or video was found to remove some of the problems of latent (or forgotten) documentation of changes to the system. This is made possible since the thoughts, ideas and changes to the system can be recorded unobtrusively as the maintenance change proceeds.

- ***Interesting environment***: the mix of multimedia information types whilst not only proving to be effective in terms of providing a range of high, medium and low level documentation about a system, also brings the maintenance process 'to life' through making the environment more interesting and by even being able to present 'in person' past maintainers.

- ***Accelerated start***: through provision and subsequent selection of the partially or fully completed generic, tailored and specific proformas, associated component ID forms, change history logs etc. an accelerated start to the comprehension activity is made possible.

- ***Consistent & traceable***: provision of a defined series of activities and stages means that the maintenance process can become more consistent and complete in terms of the activities undertaken. Additionally, when coupled with the change request and tracing sheets the status of the product becomes more clearly visible and when coupled with the review sheets, status of the maintenance process itself is made visible and traceable and hence more rigorous for audit requirements.

- ***Management & technical support***: the environment and tools provide both management and technical support for the maintenance process.

- ***System comprehension***: the approach adopted by the ISCM process which stresses the interactions of the application component parts with their environment means that a shift occurs from the comprehension activity being viewed predominantly at a program level to that of a system level. This is considered essential for today's applications which are often intimately bounded to their environmental counterparts.

- ***Solid foundation***: from an overall perspective the ISCM process, ISCM model, PISCES method and M^4 system have been shown to provide a solid foundation on which to base the future maintenance / reverse engineering of an application system.

For the reasons described above is felt that the evaluation approach adopted was appropriate. Certainly, the continuous feedback into the model, method and tool during the research proved to be invaluable. Resultantly, this has reduced the risk of having to make significant changes to the process, model or system late in their development. The range of test-bed systems were modelled successfully including those requiring linkage to third party elements and multimedia attributes. The ability model on an incremental basis to enable full or partial modelling of system was also demonstrated. This ability to comprehend the varying complexities of the test-bed applications may be seen as proof that the ISCM process, ISCM model, PISCES method and M^4 system can be used in an integral and innovative way to assist the comprehension of 'real-world' applications.

There are a number of ways in which the evaluation could have been improved or more formalised [221, 222], although it was difficult to extend to these within the timescales of the work. For example, a wider range of application types, such as those of an embedded, real-time or parallel nature could have been modelled. Additionally, the process is as yet unproven for very large industrial scale applications. However, it is thought likely that, due to the ethos of the approach and implementation via an extensible meta-CASE framework, scaleability of the ISCM process and method to industrial levels applications are not a major issue. However, complementary work in the areas of large-scale storage, indexing and version control of complex data structures containing a high proportion of multimedia related information will need to be investigated further [175, 404].

The process as it currently stands is also largely 'human-driven'. Indeed, due to the highly cognitive nature of the program comprehension and maintenance processes, it is likely that a degree of human involvement will always be required. However, the use of techniques such as neural networks, fuzzy logic and autonomous agents coupled with an increasingly populated ESIKB may enable elements of intelligent comprehension and maintenance to be feasible. These issues are discussed further in Chapter 8.

There was also a degree of subjectivity over the metrics chosen as measures of the complexity of the test-bed applications. These were deliberately kept simple for the purpose of evaluation within the scope of the thesis. It is felt that although the metrics provided were simple they were an effective measure of a number of key features by which application systems may be distinguished from each other in terms of level of complexity, i.e. lines of code, number, heterogeneity and coupling of modules. Lower level program metrics could have also been taken into account, for example measures of program control flow, operator : operand ratios, nesting levels of code etc.. However, metrics at this level were deemed inappropriate for this study which is not looking at program comprehension *per se*, but at system comprehension at a PITE level.

The M^4 system itself is also currently, very much a 'prototype demonstrator' rather than a rigorous fully developed system. It has however been appropriate for its intended purpose of supporting the research ideas developed within the thesis [30, 346]. As a result, some elements of the system and ideas stemming from the research are now being developed directly into commercial products, whilst others are being incorporated within a number of related developments.

7.3 Summary

This chapter has progressively evaluated each of the objectives identified in Chapter 1 of the thesis. In connection with each objective a brief summary has been given of why the objective was included, the work undertaken to achieve the objective; how the approach taken incorporated, built-upon or extended relevant work already in the field. The advantages offered by the results of this work and any areas requiring further development or more detailed examination have also been discussed. With regard to this, it can be stated with confidence that each of the objectives set has been successfully achieved and with positive results. However, as the nature of research is one of continual inquiry and feedback there is always scope for enhancement of this work, particularly when the research conducted has drawn together a broad range of areas as this thesis has done. In summary it can be said that a solid foundation has been established, to address more efficiently and effectively the maintenance of legacy software systems, that can be utilised immediately but which also forms the foundation for further work in this area and in associated fields.

The work within this thesis has incorporated a number of important areas of investigation, culminating in the development of a new process, model and method to address issues pertaining to the comprehension of legacy software systems. This has been achieved via the tailored application of SCM principles and techniques to the maintenance process. That this is an important area for research is supported by the findings of a study into an analysis of the industrial needs and constraints of organisations, conducted within the framework of the ESF/EPSOM project. [170]. Among the key findings of this study were that although a consensus view existed of the need for well defined and formalised procedures for maintenance, the actual formalisation of these procedures was generally much poorer than for those of the development process. The needs of maintainers covered both those of a management nature to plan, make visible and track the maintenance activity and those of technical guidelines and tools to support the activities of maintainers themselves. In particular, mention was made of the need for integration of configuration management with maintenance, including the ability to address multi-site and multi-machine configuration management [170]. Process modelling was identified as being the first step in this direction. Within this evaluation chapter, the ISCM process, its associated model, the PISCES method and the M^4 system meta-CASE framework have been shown to address all of these needs and many of those identified by the International Organisation for Standards within the SPICE (Software Process Improvement and Capability Evaluation) initiative [308].

This chapter has reviewed each objective in isolation. Chapter 8 draws together the work summarises its achievements and proposes a number of areas for future study.

Chapter 8

Conclusions and Further Work

If study's gain be thus, and this be so,
Study knows that which yet it doth not know

Shakespeare Love's Labour's Lost, Act I, Scene I

8.1 Introduction

The previous chapter discussed the work carried out in relation to each of the objectives set at the start of this thesis. The results of the research within each objective were reviewed and the degree of success in meeting each one evaluated. It is the aim of this chapter to look more globally at the success of the work and to draw final conclusions regarding the state of software maintenance today and the contribution of this research to advancing that state.

Whilst a significant amount of work has been achieved within the context of this thesis there will always be the need to explore some of the issues in greater depth and to expand the research into new areas to fully encompass the ever expanding software industry. Additionally, other areas of complementary research became apparent during the work, which although closely related were considered to be outside the scope of the thesis. These are topics for future research projects, some of which are already underway, whilst others are forming the basis of project proposals. These areas for further work are outlined in Section 8.4.

8.2 The State of Software Maintenance

Software maintenance is a discipline that has advanced considerably over the past decade in terms of its social acceptability, its management practices and its technical approaches to the effective and safe maintenance of software systems. Much of this has been aided by advances in software development practices such that maintenance is now considered from the outset thereby reducing the burden of future maintenance expenditure. For example, adoption of software configuration management tools and techniques throughout development followed by their continued use during maintenance enables the product to stay within control and to be maintained with the minimum of program comprehension effort.

However, there are also many other computer systems requiring maintenance that have been developed in an ad-hoc and undisciplined manner. These systems have many problems associated with them and as such require a significant proportion of program comprehension time, if they are to be safely maintained. This is particularly true for legacy software systems, some of which were developed well over a decade ago and were not expected to still be in operation today. Legacy systems are often characterised by immense maintenance expenditure due to the problems of their undocumented complexity and the unknown impact of changes on the equilibrium of the system, prior to the actual implementation of the change. However, legacy software is often far too valuable an asset to be thrown away and redeveloped in its entirety . This is due not only to the economic unfeasibility of developing a new system, but is also attributable to the amount of domain knowledge locked within a system that could never be fully recovered and re-implemented within a new system development.

There is thus an identified need for the ISCM process to bring legacy systems back under configuration control such that their configurations are known in terms of the individual components and how these components react with each other and with their primary and secondary environments. The resultant effect of this process is a platform for safer and more cost effective maintenance of a system. Alternatively, if the structure of the system has degenerated to such an extent that it has essentially become unmaintainable, the process establishes a solid foundation for the partial or full reverse engineering of the system.

The need to identify a system configuration is particularly pertinent at the present time due to the rapid approach of the new millennium. Not only do computer systems have to be maintained with respect to the usual problems, adaptations and enhancements, but there is also the element of risk and uncertainly as to how a system will react to 1 January 2000 [185]. Indeed, even systems that have considered the impact of the new millennium will be 'searching for' and using modules for date conversions etc. perhaps for the very first time.

Therefore, whilst considerable inroads have been made into solving maintenance problems over the past decade there is still a vast number of issues that need to be addressed. As mentioned above, one of the most critical is the time spent comprehending a system in order that only safe changes are made in spite of the complexity of its configuration and the interrelationships between the component parts. Models are starting to appear that exclusively address the maintenance aspect of the software lifecycle, however, there is still a need to refine these models and define within them a finer level of activity granularity. The effect of this will be to assure process repeatability and consistency within the areas of program comprehension and software configuration reclamation across the global maintenance community.

The main aim of this research has therefore been to provide a solution to the maintenance problems of legacy software systems through the definition of a new process, *Inverse Software Configuration Management (ISCM)*. ISCM can significantly reduce the time required for program comprehension and can assist in bringing existing software systems back under control. Section 8.3 together with Figure 8.1 summarises the progression of the work undertaken within the scope of the thesis, and highlights the original contribution that has been made by this research.

8.3 Overall Summary of Work Undertaken

To provide the underpinning knowledge for the concepts within this thesis a general review of the current state of software maintenance and its associated disciplines of software configuration management, reverse engineering and redocumentation was undertaken. This identified the need to investigate a solution to the high program comprehension overhead associated with maintenance of legacy software systems.

The area of process modelling was then investigated in more detail to enable a new process, Inverse Software Configuration Management (ISCM) to be defined and placed within the framework of existing software maintenance process models. This process focuses on the reidentification and subsequent control of the components and configurations of legacy software systems. A study was therefore subsequently undertaken to establish the different types of components associated with a software system and the nature of their inter-relationships. This led to a number of software configuration abstractions being defined, thereby enabling a software system to be studied on an aspect-by-aspect basis prior to building up the complete configuration description of a software system. This information together with the set of rules for reclaiming the configurations was stored within the Extensible System Information Base (ESIB) through the definition of an Inverse Configuration Description Language (ICDL).

In order to assist the consistent and coherent use of the ISCM process across a wide range of software applications and system architectures, the PISCES (Proforma Identification Scheme for Configurations of Existing Systems) method was developed as a defined series of procedures and guidelines. To underpin the method and to offer a user-friendly interface to guide the configuration information collection and collation activity a series of templates, the Proforma Increasing Complexity Series (PICS), was developed. The purpose of the PICS is two-fold, in addition to data collection, it also serves as a reporting mechanism for the progressively reclaimed system configurations.

It is essential to be able to readily document and retrieve information if it is to be usefully employed on large and complex projects. The PICS series was therefore developed for on-line usage and as such is maintained as part of the PISCES M^4 prototype system. The M^4 system was developed as a meta-CASE (Computer Aided Software Engineering) environment to drive and semi-automate the ISCM process. Provision of a meta-CASE environment rather than a rigidly defined CASE tool has enabled maximum generic modelling capability. Additionally, the meta-approach is very cost-effective in that it offers increased opportunities for exploitation of the host system environment tools. Of particular interest within the M^4 system is the provision of a Multimedia User Interface (MUI) to the maintenance process.

As a means of evaluating the PISCES method and in order to provide on-going feedback into the ISCM process model, a number of practical applications were studied. These practical applications also enabled the features and facilities of the M^4 prototype system to be assessed.

This research has thus considered a number of concepts, some of which are innovative in nature themselves and others of which are used in a novel manner. In combination these concepts may be considered to significantly advance the knowledge and understanding regarding program comprehension during the maintenance of legacy software systems. These concepts are summarised in Figure C8-1 and the contributions of this work to the knowledge area may be summarised as:

- ***Definition of the ISCM process***: an innovative process which adapts the traditional activities of the development-driven SCM discipline to the requirements of the maintenance-driven program comprehension process.

- ***Development of the ISCM process model***: a new model that integrates the ISCM activities into the software lifecycle thereby enhancing the current modelling of the software maintenance process. Although emphasis is strongly placed on the identification activity, the process also supports the change control, status accounting and audit functions of SCM.

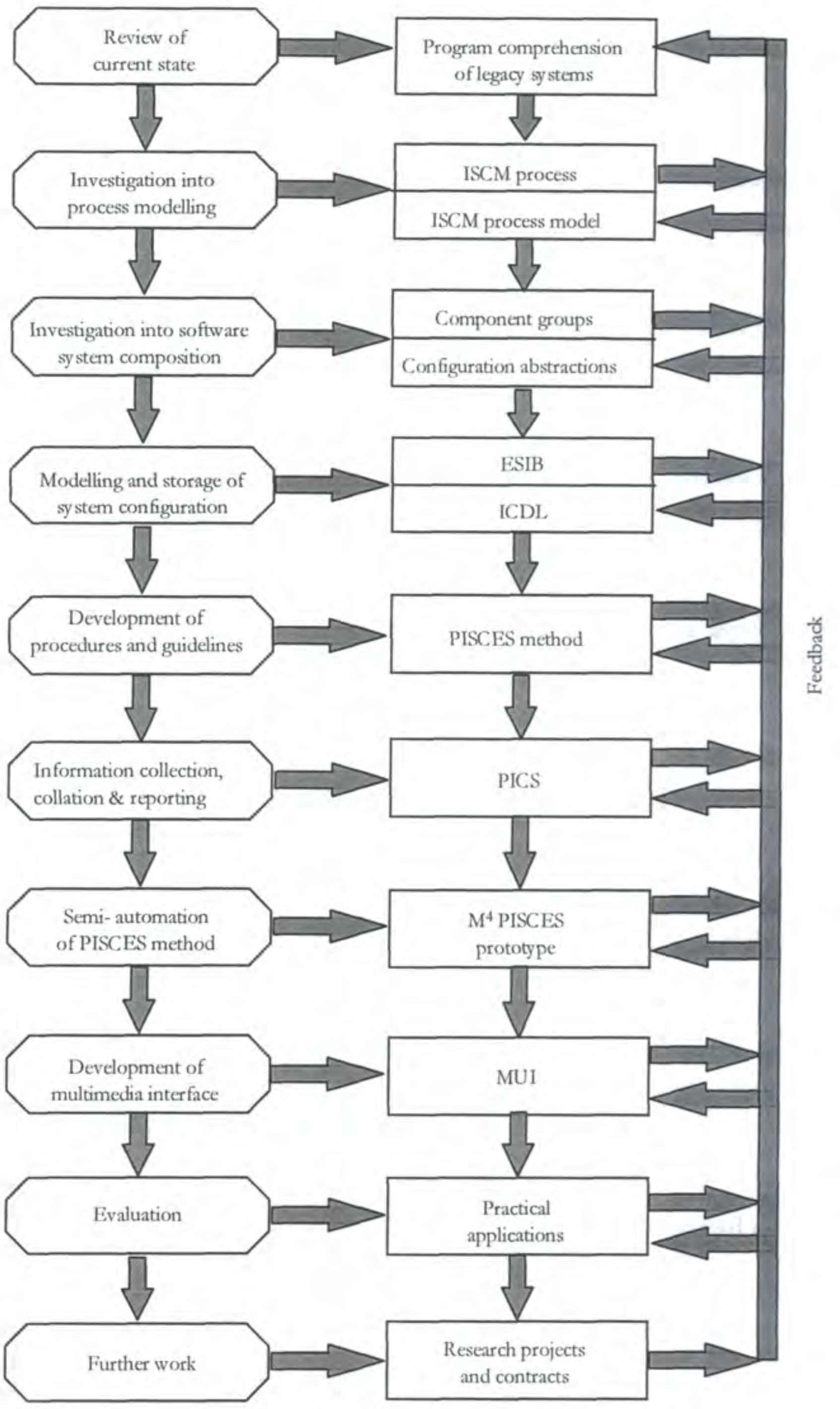

Review of current state
Program comprehension of legacy systems
Investigation into process modelling
ISCM process
ISCM process model
Investigation into software system composition
Component groups
Configuration abstractions
Modelling and storage of system configuration
ESIB
ICDL
Development of procedures and guidelines
PISCES method
Information collection, collation & reporting
PICS
Semi- automation of PISCES method
M⁴ PISCES prototype
Development of multimedia interface
MUI
Evaluation
Practical applications
Further work
Research projects and contracts
Feedback

Figure C8-1 Summary of research

- ***Identification of key component groups and configuration abstractions***: a more comprehensive approach to the modelling of software system architectures than in many previous methods. The approach enables abstractions to be developed of the configuration descriptions. These may range from a simple model of the core application baseline system to the complete but often incomprehensible entire system configuration incorporating all application, program family and environmental component parts.

- ***Development of the ICDL and ESIB***: a configuration description language that specifically addresses the modelling of components and relationships prevalent in legacy systems. This language adopts an innovative approach to the modelling of software systems through the introduction of the concept of programming-in-the-environment (PITE) which extends the previous work in the field of programming-in-the-large (PITL). The structure of the ICDL forms the basis of information storage within the ESIB. The ESIB has the flexibility to incorporate documents or references to documents of all media types.

- ***Development of the PISCES method***: a new method that provides the guidelines and procedures for implementing the ISCM process. The method addresses both technical and managerial aspects of the SCM process. Technical aspects include information reclamation and storage mechanisms, while the managerial aspects encompasses the consistent and complete documentation of components and configurations and the change status of these items within an application.

- ***Definition of the PICS***: a natural language representation of software system architectures suitable for use by users, managers and maintainers. The PICS drive the incremental information collection, documentation and reporting activities. Although the use of proformas or templates in themselves is not unique to this research, they form an integral part of the ISCM process. Hence the context in which they are used is considered innovative. The templates or proformas enable abstraction in terms of generic, tailored and specific modelling of software system architectures. The proformas also contribute to the control of the maintenance process, by their recognition of the increased heterogeneous nature of systems due to the widespread uptake of multimedia and web based products, and the need to document and control such systems.

- ***Genetic fingerprinting of a system***: the progressive abstraction in terms of complexity of modelling afforded by the PICS eventually leads to '*genetic fingerprinting*' of a software system application. This genetic fingerprint represents a software system configuration at a determined maintenance point in time and can therefore be used as a baseline on which to measure further changes to, or evolution of, the system. Cumulatively over time these 'genetic fingerprints' can document the maintenance history of a system at an entire configuration rather than being restricted to a component-by-component basis. Additionally, as generic aspects of a system can be modelled and subsequently refined the method is suitable for a wide range of systems types across a substantive range of operating platforms.

- *Design and implementation of the PISCES M^4 meta-CASE environment*: this prototype set of tools provide an environment for the management and control of the entire ISCM process. In particular, it enables on-line implementation of the proformas which drive the data collection and reporting process. The framework also enables the flexible integration of host resident tools, techniques and methods pertinent to a particular application or system type.

- *Development of the MUI:* implemented as part of the M^4 system, this concerns the development of a Multimedia User Interface (MUI) to the program comprehension process. Graphical front-ends to maintenance products are now reasonably well established with their integration of text and diagrams. However, the M^4 system extends the user interface to enable a richer set of knowledge and information to be recorded for a software system undergoing maintenance than is possible with text documents alone. Similarly, the report generation facilities are also enriched through the application of video, audio and animation, adding value to the traditional text and graphics environments. Additionally, although several maintenance products incorporate hypertext links between and within documents, the M^4 system extends this to provide hypermedia links between and within the different media types.

8.4 Further Work

The time-scale for the research dictated that not every idea could be investigated to the same level within the context of this thesis. A number of future developments of the work have been identified. Some of these areas involve enhancement and continued development of the work produced within the thesis; others involve exploring complementary areas of research. Some of these project ideas are already funded and underway whilst others are being incorporated into project proposals. These developments are discussed in the remainder of this chapter.

8.4.1 Proving and Refinement of the ISCM model

Software maintenance is a still a new discipline, particularly when compared with many other engineering disciplines which have had centuries to become established in their working practices and which are not subject to such an explosive rate of change in technology as that currently evident in the computer industry [322]. There is therefore still considerable flux within the methods and tools being produced [28].

The ISCM process and its associated model, PISCES method and tool have been designed and developed within the context of this thesis. Whilst initial studies show that ISCM has the potential to benefit the maintenance community in terms of assisting the program comprehension process and thereby bring a system back under configuration control, there is still a number of extensions and enhancements that can be made to the model. Developments are currently being undertaken in the following areas:

- ***Refinement of the model and method:*** through detailed examination of applications similar to those used within the thesis, but with increased numbers of components and levels of complexity.

- ***Extension of the model and method:*** through the detailed modelling of the special characteristics of distributed, real-time, multimedia and embedded systems. Associated with these different application types, is also the requirement to model the cognitive aspects of the corresponding domains of which the systems form a part or in which they are intended to operate. Modelling of these domains could even be extended to incorporate wider organisational activities such as marketing, sales and user support if these were felt to impact significantly on the software system and the process by which it is maintained [241].

The above refinements and extensions may manifest themselves as changes to the activities encompassed within to the ISCM process model; modifications to the way in which these activities are implemented through the PISCES method; or amendments and additions to the number of fields and attributes that must be included in order to progressively populate the PICS. There is also the need to examine further the degree to which the PICS and the ICDL can be populated, stored and maintained concurrently.

8.4.2 Formalisation of the PICS Series

The PICS approach to modelling provides a useful representation of software system architectures in terms of natural language understanding of individual components, their location, descriptions and combination into software configurations. This is very relevant in terms of enabling users, managers and maintainers to gain an overall understanding of the system and its environment. However, this form of representation can be lengthy and, as with all natural language representations, occasionally ambiguous in its meaning. There is therefore a perceived need for investigating a more formal approach to the modelling of the software system descriptions defined within this thesis through the use of semi-formal object-oriented (O-O) techniques and rigorous formal method languages either singularly or in combination [281].

- ***Object-oriented techniques***: modelling for maintenance using object-oriented (O-O) techniques is becoming an increasingly viable approach considering the paradigm shift during development, away from the traditional functional approach towards that of an object-based or object-oriented nature. Indeed, even the most conservative forecasts consider that by the year 2000 more than half of all systems will be based on the O-O approach [192]. The impact of this can be seen as twofold: systems embracing the O-O paradigm should exhibit characteristics of increased maintainability and, in theory at least, it should be more intrinsic to model and maintain O-O systems using O-O approaches.

  Object-oriented technology is an approach that models systems around real world concepts encapsulating both their functionality (behaviour) and data into 'objects' or object types. Also central to the O-O paradigm is the notion of abstraction and classification in order to reduce the complexity of a system. If a software system is considered as being a collection of definable components, exhibiting definable behaviour, having definable properties and communicating with other definable groups, the use of O-O technology to model software system architectures seems to be a natural progression. This is especially true given that the ISCM approach is already heavily biased towards the principles of abstraction in terms of its component group view of software configurations and its use of the PICS to model abstractions of the system.

- ***Formal Method Languages:*** Although an object-model would go some way towards formalising the information extracted by the PISCES system relevant to the description of software system architectures being maintained, there is still room for confusion and misinterpretation. Therefore, an approach now being considered is that of translating the proforma information into a mathematically based formal language. By doing so, a precise and unambiguous specification of the state of a system under maintenance could be provided. This has the added advantages of formalising the work, providing the basis for traceability and proof of correctness and providing a basis for reverse engineering of the system since they could assist in design and specification rediscovery for the system. Indeed, if the original specifications were available it may be possible through formal methods to detect inconsistencies, omissions and ambiguities in the system being maintained. At the very least ambiguities for future maintainers would be removed.

The formality of the object-oriented and formal method types of representation may thus be able to more succinctly describe the software configurations. These representations could also act as a useful shortcut once a maintainer had grasped a good general understanding of an application system through the more visually developed PICS series. Additionally, such representations would present a more rigorous method for verifying the consistency and completeness of any subsequent changes to the configured system.

8.4.3 Pattern Oriented Software Maintenance (POSeM)

Pattern architectures are currently the subject of a growing body of research. They capture well proven experience in software development and help promote good design practice [67, 68]. However, to date they have concentrated on the developmental aspects of the software lifecycle. It is the intention to develop a comprehensive series of patterns which can form the basis of the software maintenance process. Pattern definition and usage could be explored in the areas of reuse, redocumentation, and reverse engineering, and at the code, design, specification or even organisational levels [67, 107, 217, 273, 279]. Patterns may also be investigated as a way of providing procedures for defining and guiding the maintenance process; as a means of documenting legacy software architectures at varying levels of abstraction and for different domains; and for providing a common vocabulary and understanding for maintenance principles and techniques. A further use of patterns may be their use in combination with formal methods to ensure preservation of system functionality during the re-engineering process [229].

8.4.4 Extension of the ESIB

Underpinning the model and method is the Extensible Systems Information Base, ESIB. This is the storage medium for the rules and reclaimed information about a system type or configuration. There are a number of directions in which this area can be investigated further:

- ***Extension of the Rule Base***: gradually as more applications are maintained and modelled using the ISCM process, more information and implicit and explicit rules for extracting this information can be built into the knowledge base. This information may be of both of a specific nature with regard to a particular application or of a more general nature pertaining to software systems in general. As a consequence of this continual influx of information the role of the ESIB will gradually build up to that of a '*maintenance oracle*'. This has a number of potential uses in terms of increasing the ability of the M^4 system to 'intelligently' maintain an application system, or to automate the process of maintenance and thereby evolve to become an Extensible System Information **Knowledge** Base (ESIKB). With this in mind the application of techniques such as case-based [11], model-based [285, 347] and functional [382] reasoning should be investigated with respect to the software maintenance process.

- ***Object-Oriented Representation of the ESIB***: during the course of the research the database component of the information base (database + knowledge) has moved from a flat-file representation to that of a relational implementation. However these implementations have purposely been limited to the storage of references to components and their configuration descriptions rather than storage of the actual components themselves. Physical storage of the component themselves has been handled by the underlying file storage mechanism of the host computer. This is not necessarily a practice to be changed, since such an approach retains maximum system flexibility and portability (particularly with a flat-file representation) and avoids the often considerable difficulties associated with storage of large documents and files with complex internal structures. This issue is becoming particularly relevant when considering the multimedia and distributed nature of many of today's software systems. It also means that the concept of delta storage of different component versions can be handled by whatever version management system has been incorporated into the M^4 toolset and environment.

  However, advances are rapidly being made in the area of extended-relational and object-oriented database systems which are far more suited to the storage of complex, structured and multimedia objects. This area therefore warrants investigation, particularly since part of the ISCM process is already concerned with the object-oriented and pattern modelling of software system architectures. There are currently a number of object-oriented databases appearing in the market place and being developed within the research arena whose facilities would be useful to investigate [78, 219]. Such systems include O_2, Ontos, Iris, Orion and Gemstone [106, 219, 397]. However, many of these databases have yet to be proven in widespread large industrial situations.

  Investigation into the structure of databases for handling multimedia attributes can also be used to further develop the M^4 system towards that of a computer supported co-operative work (CSCW) environment. These developments could potentially make use of a software layer to translate web requests for information into SQL queries, and generate web pages dynamically to display results to the user [129].

8.4.5 Further Development of the M^4 Meta-CASE Environment

There are plans to further extend the facilities and features offered by the PISCES M^4 system. Many of these are already underway and may be considered under the following headings:

- ***True hypermedia linkage***: it is intended to extend the hypermedia capabilities of the PISCES M^4 system. Currently, full use is made of multimedia attributes, however, in the main, linking and control of related items of data is still implemented through the control centre. However, hypertext linkages (text:text) between and within textual documents have been implemented, as have hypermedia extensions (media:media) from within text documents to the other media types. The next version of the M^4 system will fully implement hypermedia extensions within and between all media types. This much finer granularity of linkage will enable a more elegant and targeted coupling of audio, video, graphics and text documents, thereby further increasing the effectiveness of the program comprehension process. The Hypermedia Browsing and Editing Facility (HyBrEF) has already been developed to address these issues but has not yet been implemented within the M^4 framework.

- ***Further support for ISCM of multimedia and distributed products***: applications that make use of multiple media types are fast becoming the norm. It has been stated that due to technical demands, large heterogeneous, networked and distributed multimedia systems, need new object-oriented, user friendly software development tools as well as tools for retrieval and data management [175]. The PICS and Mu^2PITS work has addressed many of the issues surrounding the data management aspect of such systems particularly with respect to documenting the component and configuration identification and change status of both standalone and distributed multimedia products. However, there is still scope to enhance this work through the inclusion of features such as [125]:

  - ◊ Automatic prompting for updates to components affected by changes to other component through linkages in their dependency relationship graph.

  - ◊ Addition of a customer/client database for recording client-specific multimedia product configurations.

◊ Explicit rather than implicit linkages between the Mu^{2}PITS system and the component files to enable closer linking of identification and change information to each specific component.

◊ Linkage to an e-mail facility for faster communication regarding the state of a change and for automatically informing other users who might be affected by the intention to make a component change.

◊ Automatic detection of file formats and subsequent highlighting of possible configuration conflicts, for example if trying to configure a product from a combination of PC, Mac or Sun system components.

◊ Adaptation of the Mu^{2}PITS system to control the flow of information in a web page in terms of documenting the hyperlinks.

◊ Technical evolution of the M^4 system with regard to issues such as improved query processing, transaction, buffer and storage management, and recovery and security [175]. It is also intended to investigate the incorporation of speech recognition as a means of enabling the maintainer to interact with/invoke the different parts of the system [326].

- ***Addition of tool base:*** the framework of the M^4 system means that more program comprehension and data extraction tools can be added to, or exchanged with existing elements in the toolset. This provides for the most flexible and system geared framework, which is a positive feature for CASE tools which have been reported as suffering from a lack of flexibility in the past [327]. Additional tools will enhance the ability of the system to define new rules and add increasing amounts of knowledge to the ESIB.

- ***Movement of the ISCM environment into the CSCW Arena***: work is already underway to extend the use of the PISCES M^4 system into the Computer Supported Co-operative Work (CSCW) arena. CSCW is a developing field of information technology which builds on the capability for communication emerging from software developers and network providers [94, 176, 304, 349]. CSCW is extending the boundaries of the more established Groupware [157, 246] concepts which cover software support for groups of workers, ranging from e-mail at its simplest to distributed real-time video conferencing at its most complex. The potential of extending the M^4 system to make use of teleconferencing techniques over the Internet to allow distributed software development and maintenance is of great commercial interest to many companies, particularly with the increasing encouragement of teleworking [248] and the need to maximise experienced maintenance staff availability. Eventually, the stand-alone M^4 system will evolve to become a multi-user distributed networked system across the Internet.

- *Virtual Reality Maintenance*: ultimately, the M^4 system environment will be augmented through the use of an advanced 3-D network based Virtual Reality (VR) interface. Such a system will enable the maintainer to 'walk around' a software company and to discuss activities with other maintainers who are present on the network. Indeed, present day maintainers will also be able to roam around the system archives and 'interact' with virtual software developers and maintainers from the past, captured for posterity within the environment through the implementation of hologramic images. The VR interface coupled with storage of ever increasing amounts of knowledge about a system will advance the concept of the maintenance oracle. This will enable the problems of the development-maintenance / maintenance-maintenance time-space divides and the loss of key personnel to be largely overcome.

8.4.6 Further Automation of ISCM Process

Essential to the ISCM process is the collection and reporting of information regarding the software system architectures. Currently this is achieved through a combination of manual and semi-manual procedures but which have the potential to be more fully automated. Some of these areas are outline below:

- ***Automated retrieval of information***: currently, the ISCM process makes use of host-resident information extraction tools such as awk, grep, make etc. However, there is still a high proportion of work which has to be done manually, albeit through interactive prompting by the system and through the use of the PICS templates which can be saved and incrementally built up. Addition or development of further and more specific tools for data extraction will enable a higher degree of automation of the process.

- ***Progressive population of the ESIB***: further automation of the information extraction and PICS completion process, will be achievable through the continual build-up of the ESIB over a period of time. This is due to the progressive population of the information knowledge base with rules for generic system modelling and with historical data pertaining to specific applications requiring comprehension. Gradually as the ESIB is built up, patterns of behaviour of particular system types being maintained will become evident increasing the degree of automation of the process. Additionally, the PICS series for specific applications undergoing maintenance will gradually become more complete and hence the starting point for comprehension will be markedly advanced.

- ***Automation of the translation process:*** there is also scope for investigating the automatic generation of semi-formal and formal representations of the system configurations. This may be achieved through the development or incorporation in the M^4 toolset of utilities for more effective parsing of the system description document and of specific diagramming utilities for outputting the results. Automation in this respect will not only make the process more efficient but will assist in the verification process of changes made to system configurations and in defining the baselines at specific points through the maintenance process.

8.4.7 Intelligent Program Comprehension

As stated in the previous section the PISCES method supports a degree of process automation, and a number of areas for further automation have been identified. However, there is still a key requirement within the program comprehension process for a human maintainer to drive the information collection and collation process as well the interpretation of the results. Additionally, the quality of the decisions regarding a system is very much influenced by the prior knowledge, experience and expertise of the past and present maintainers of the system. In particular the ESIB accepts information directly from 'experts' in the field or collates it as a result of rules defined by these experts. For an 'inexperienced' maintainer it may be difficult to assess with confidence the quality of the decisions made or the accuracy of the information entered into the ESIB. Additionally, although hypermedia can be viewed as a powerful tool for building software modelling support tools, it lacks the reasoning or inference ability to generate a truly 'intelligent' [79] or 'cooperative' [86, 196] system. Advances being made in the field of Artificial Intelligence (AI) could however be exploited to provide elements of intelligence within the program comprehension process, thereby advancing the degree of auto-maintenance possible. The technologies that appear to afford the most potential in this area are the application of software agents, neural networks and fuzzy logic:

- ***Software agents:*** these are self-contained programs which are capable of controlling their own decision-making and accomplishing tasks for their user through actions based on their perception of the environment [206, 296]. Typically agents exhibit behaviours of autonomy, social ability, responsiveness, pro-activeness [288], persistence and communicativeness [232], although this is determined to some extent by the type and nature of the agent. Agents may be collaborative (interact with other agents); mobile (roam wide area networks); information (manage, manipulate and collate information); reactive (respond to state of the environment); and hybrid (combinations of two or more agent types) [288].

- ***Neural networks:*** present a completely different paradigm for solving problems than that offered by conventional computing based on well defined numerical algorithmic processing. Neural networks encompass networks of adaptable nodes which, through a process of learning from task examples, store experiential knowledge and make it available for use [391]. Indeed, many of the basic functions provided by neural networks mimic the abilities of the human brain. These functions include those of classification, clustering, associative memory, modelling, time-series forecasting and constraint satisfaction [47]. Many of these areas which rely on attributes such as creativity, generalisations and understanding have proved to be difficult areas for conventional computer processing. Neural networks however, have the potential to infer and induce from what might be incomplete or non-specific information. They can also learn appropriate modes of behaviour particularly when presented with real-world data. As such the network can be taught not only to recognise particular patterns of data as they occur, but also to recognise similar patterns by generalisation [391]. The increased profile of the abilities of neural networks have led to predictions that the market for such networks will grow world-wide at a rate of 46% per year [215].

- ***Fuzzy systems:*** this is another area which although studied for many years is only just becoming popularised as a technique for modelling uncertainty in a range of system applications. Fuzzy systems are concerned with imprecision and approximate reasoning [216] and through the enabling techniques of fuzzy logic and fuzzy sets can be used as a method for reducing as well as explaining system complexity. This modelling of systems using fuzzy logic is based on linguistic rather than mathematical representation of data and processes. Indeed it has been stated by Zadeh, that as the complexity of a system increases, the ability to make precise and significant statements about its behaviour diminishes [112]. Such a phenomenon therefore also diminishes the ability of traditional mathematics to represent the real-world characteristics of software systems and their architectures. However through the use of fuzzy logic, benefits may be shown to accrue in the following areas: the ability to model highly complex business problems; improved cognitive modelling of expert systems; the ability to model systems involving multiple experts; reduced model complexity; and improved handling of uncertainty and possibilities [112].

There are a number of ways in which the intelligent learning of software agents, and the soft computing techniques associated with neural networks and fuzzy logic may be used either in isolation or in combination [155, 216] to increase the intelligence and decision making of the M^4 environment and thereby the effectiveness of the program comprehension process. The areas for investigation include:

- ***Automated modelling:*** the ISCM process has made use of modelling techniques based on simple set theory. Whilst this has provided a convenient mechanism for modelling the component groups and their composition into viable configurations, there was often a degree of uncertainty during the modelling surrounding membership of the component groups. Definition of fuzzy configuration sets for modelling the configuration abstractions therefore warrants further investigation as a more exact real-world representation of the configuration composition process.

  Additionally, many of the benefits accruing from reducing and explaining system complexity may be directly related to the maintenance of software systems enabling reduced mean-time-between-failure; improved mean-time-to-repair; easier and more stable extensibility of existing systems; and improved understandability.

- ***Automated data extraction:*** currently the information placed in the ESIB and extracted for reporting purposes can be done on a semi-automatic basis. However, the placement or extraction must be driven by the maintainer, either via selection from a defined set of queries or by construction of a query specifically adapted to current information needs. It would be useful if part or all of this process could be conducted automatically and presented to the maintainer prior to them making the change during the program comprehension process. Intelligent software agents are a mechanism whereby this could be achieved. Hence the use of agents for the collection and collation of the information pertaining to the software system configurations and the subsequent population of the PICS and ESIB is being considered [13].

Software agents at their most basic level may be programmed with a set of rules to enable them to 'wander around' a standalone system and to gather pre-programmed matches of data requirements [251]. Intelligence can be added to such agents through artificial evolution such that agents can gradually evolve their degree of autonomy until they are capable of meeting user needs and adapting to the changing requirements of their users without the need for reprogramming [226, 251]. Additionally, to substantially advance the program comprehension process for large heterogeneous systems, multi-agent approaches based on the distributed problem-solving paradigm and distributed artificial intelligence should also be employed [277, 367]. This approach is known as co-operative information gathering and involves concurrent, asynchronous discovery and composition of information spread across a network of information servers. Indeed agents are being evaluated as a means to support legacy systems in the area of kiosks for telephone sales and support and information access for service technicians [232].

- ***Data mining for maintenance:*** data mining is the efficient discovery of valuable, non-obvious information from a large collection of data [15, 47]. That is, it centres on the automated discovery of new facts and relationships in data. As more systems are maintained through the ISCM process and the amount of data held within the ESIB increases it may become increasingly difficult to find and select the data relevant to a particular application, and to ensure that the host systems has been completely searched for relevant configuration components. There is thus scope for investigating more intelligent data search mechanisms and for the creation of data warehouses for program configuration data. Again techniques such as software agents, neural computing and fuzzy logic have significant roles to play in this area.

- ***Intelligent program comprehension & auto-maintenance:*** it is intended to investigate a shift towards intelligent program comprehension whereby the M^4 system would not only be able to automatically identify the characteristics of a system configuration but through analysis of these characteristics could also flag up trouble spots in the system meriting special attention during the maintenance process.

The use of software agents, given a set of fuzzy logic rules and learning through the application of neural networks may also enable a degree of auto-maintenance to take place. For example, in networked based systems, it may be possible for program comprehension and problem diagnosis to be performed remotely from a central location. The M^4 system could incorporate a number of software agents that have the responsibility for storing, controlling and monitoring information about each part of the system. The application of this technology has been used in network diagnostics for a number of years [71] and hence there is the potential to extend these ideas towards the management of the software is-itself rather than limiting diagnostics to the hardware components of a distributed network [218]. It is envisaged that through agent monitoring of metrics regarding factors such as component stability, age, size etc. and comparing these to fuzzy data sets or rules it may be possible to make predictions of the need to perform maintenance on certain parts of the system.

- ***Intelligent decision support, cost-benefit, and risk analysis for reverse engineering:*** allied with the issues of intelligent program comprehension is application of artificial neural networks in order to determine the cost-benefits and/or risks of continuing to comprehend or maintain all or part of a system [321, 354, 355]. Artificial neural networks that have been taught [130] and tested by means of experienced maintenance practitioners, could result in accurate recognition of when to reverse engineer, re-engineer or redevelop parts of a software system.

8.5 Summary

Two major criteria must be met to satisfy the requirements for PhD level research. Firstly, an area of research must be identified in which an original contribution can be made to the existing body of knowledge concerning the relevant field. Secondly, it must be shown that the research process itself can be carried out in a defined and systematic manner and the results communicated effectively to other researchers.

Very few people would disagree that software systems are becomingly increasingly pervasive and important within our everyday and working lives. Hence, there is a very urgent need, if the rapidly growing consumer and industrial products of today are to remain safe and reliable, and we are to avoid a software crisis far greater than that of the early 1980s, to retain or regain control of systems that are undergoing development and maintenance.

The aim of this thesis has been to design, develop, justify and evaluate the effectiveness of the ISCM process, ISCM model, PISCES method and M^4 system environment in order to demonstrate the contribution of the research to the software maintenance discipline. As discussed throughout the thesis, the main aim of the ISCM process is to enable a legacy software system to be brought back under configuration control through a series of defined technical and managerial activities and procedures. The ISCM process is based on the inverse application of traditional SCM concepts and a model defined at a finer level of activity granularity than in many previous process models. The ISCM process thereby enables configurations of software systems to be incrementally reclaimed and unambiguously documented via the PISCES method and M^4 system to enable safe and effective maintenance to take place.

The research and resultant ISCM process has established a solid foundation for more effective program comprehension during the maintenance of legacy software systems. The process through which the research has been carried out has been structured and iterative in nature. In summary, the problems of program comprehension during maintenance were analysed, the ISCM concept was defined and subsequently developed as the PISCES method, which in turn was implemented and semi-automated through the PISCES M^4 system before being tested with a number of practical applications. Each stage of the research provided feedback into the previous stages and any identified corrections, amendments and enhancements were encompassed to strengthen the practical applicability of the ISCM approach. A number of publications have been produced from the research and a number of funded research and commercial projects are underway, building upon the concepts developed within the thesis.

References

[1] Abbott, R.J., *Knowledge Abstraction*, Communications of the ACM, Vol. 30, p.666, August 1987.

[2] Adams, C., *Why Can't I Buy a SCM Tool?*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[3] Adler, R.M., *Emerging Standards for Component Software*, IEEE Computer, pp. 68-77, March 1995.

[4] Adolph, W.S., *Cash Cow in the Tar Pit: Reengineering a Legacy System*, IEEE Software, pp. 41-47, May 1996.

[5] Aho, A.V., Kernigan, B.W. and Weinberger, P.J., *Awk - a Pattern Scanning and Processing Language*, Software Practice and Experience, Vol. 9, No. 4, pp. 267-280, April 1980.

[6] Ahrens, J.D. and Prywes, N.S., *Transition to a Legacy- and Reuse- Based Software Lifecycle*, IEEE Computer, pp. 27-36, October 1995.

[7] Ainsworth, P., *The Millennium Bug*, IEEE Review, pp. 140-142, July 1996.

[8] Alderson, A., Bott, M.F., Falla, M.E., *The Eclipse Object Management System*, Software Engineering Journal, pp. 39-42, January 1986.

[9] Alderson, A., *A Space-Efficient Technique for Recording Versions of Data*, Software Engineering Journal, pp. 240-246, November 1988.

[10] Alkhatib, G., *The Maintenance Problem of Application Software: An Empirical Analysis*, Journal of Software Maintenance: Research and Practice, Vol. 1, pp. 83-104, 1992.

[11] Allemang, D., *Combining Case-Based Reasoning and Task Specific Architectures*, IEEE Expert, pp. 24-34, October 1994.

[12] Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D. and Posner, J., *ClearCase MultiSite: Supporting Geographically Distributed Software Development*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[13] Ambite, J.L and Knoblock, C.A., *Agents for Information Gathering*, IEEE Expert, Vol. 12, No. 5, pp. 2-4, September-October 1997.

[14] Ambriola, V., Bendix, L. and Ciancarini, P., *The Evolution of Configuration Management and Version Control*, Software Engineering Journal, November 1990.

[15] Anand, S.S., Scotney, B.W., Tan, M.G., McClean, S.I., Bell, D.A., Hughes, J.G. and Magill, I.C., *Designing a Kernel for Data Mining*, IEEE Expert, pp. 65-74, March-April, 1997.

[16] Andriole, S., *Software: The Good, the Bad, and the Real*, IEEE Software, pp. 45-47, January-February 1998.

[17] ANSI/IEEE Std 828-1983, *IEEE Standard for Software Configuration Management Plans*, In: Software Engineering Standards, IEEE, New York, 1984.

[18] Antoniol, G., Fiutem, R., Merlo, E. and Tonella, P., *Application and User Interface Migration from BASIC to Visual C++*, In Proceedings Conference on Software Maintenance, pp. 76-85, Nice, France, October 1995.

[19] Aoyama, M., *Sharing the Design Information in a Distributed Concurrent Development of Large-Scale Software Systems*, In Proceedings Conference on Software Maintenance, pp.442-449, 1996.

[20] Asirelli, P. and Inverardi, P., *EDBLOG: a Kernel for Configuration Environments*, International Workshop on Software Version and Configuration Control, pp. 422-425, Grassau, January 1988.

[21] Baalbergen, E.H., *Design and Implementation of Parallel Make*, Computing Systems, Vol. 1, No. 2, pp. 135-158, Spring 1988.

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

[23] Babich, W.A., *Software Configuration Management*, Addison-Wesley, 1986.

[24] Balzer, R., *A 15 Year Perspective on Automatic Programming*, IEEE Transactions on Software Engineering, Vol. SE-11, No.11, November 1985.

[25] Bao, Y. and Horowitz, E., *Integrating Through User Interface: A Flexible Integration Framework for Third-Party Software*, Conference on Software Maintenance, pp. 336-342, 1996.

[26] Basili, V.R. and Turner, A.J., *Iterative Enhancement: a Practical Technique for Software Development*, IEEE Transactions on Software Engineering, Vol. 1, p. 330, December 1975.

[27] Basili, V.R., *Viewing Software Maintenance as Re-use Oriented Software Development*, IEEE Software, Vol. 7, pp19-25, January 1990.

[28] Basili, V.R., *The Role of Experimentation in Software Engineering – Past, Current and Future*, In Proceedings Conference on Software Engineering, pp. 442-449, Berlin, Germany, March 1996.1996.

[29] Baumann, P. and Kohler, D., *Archiving Versions and Configurations in a Database System for System Engineering Environments*, International Workshop on Software Version and Configuration Control, Grassau, pp. 313-325, January 1988.

[30] Baumer, D., Bischofberger, W.R., Lichter, H. and Zullighoven, H., *User Interface Prototyping – Concepts, Tools, and Experience*, In Proceedings Conference on Software Maintenance, pp. 532-541, 1996.

[31] Bazelmans, R., *Evolution of Configuration Management*, ACM Software Engineering Notes, Vol. 10, No. 5, pp. 37-46, 1985.

[32] Belady, L.A. and Lehman, M.M., *A Model of Large Program Development*, IBM Systems Journal, No. 3, pp. 224-252, 1976.

[33] Ben Arfa, L. and Mili, A., *Software Maintenance Management in Tunisia: A Statistical Study*, in Proceedings Conference on Software Maintenance, pp. 124-129, San Diego, California, November 1990.

[34] Bennett, K.H., Cornelius, B., Munro, M. and Robson, D., *Software Maintenance*, in J. McDermid, ed. Software Engineer's Reference Book, Chapter 20, Butterworth-Heinemann, 1991.

[35] Bennett, K.H., *Automated Support of Software Maintenance*, Information & Software Technology Journal, Vol. 33, No. 1, pp. 74-79, Jan-Feb 1991.

[36] Bennett, K.H., *Legacy Systems: Coping with Success*, IEEE Software, pp. 19-23, January 1995.

[37] Benoit, H., *Digital Television MPEG-1, MPEG-2 and Principles of the DVB System*, Arnold, 1997.

[38] Berners-Lee, T., *WWW: Past, Present, and Future*, Computer, pp. 69-77, October 1996.

[39] Bernstein, P.A., *Middleware: A Model for Distributed System Services*, Communications of the ACM, Vol. 39, No. 2, pp. 87-97, February 1996.

[40] Bernstein, P.A., *Database System Support for Software Engineering*, In Proceedings 9th International Conference Software Engineering, Monterey, California, pp. 166-178, March 1987.

[41] Bersoff, E.H., Henderson, V.D., Siegel, S.G., *Software Configuration Management : A Tutorial*, Computer, pp. 6-14, January 1979.

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

[43] Bersoff, E.H., *Elements of Software Configuration Management*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, pp. 79-87, January 1984.

[44] Bhansali, P.V., *Universal Safety Standard is Infeasible - for Now*, IEEE Software, pp. 8-10, March 1996.

[45] Bicknell, P.A., *Software Development and Configuration Management in the Cronus Distributed Operating System*, Proceedings Conference on Software Maintenance, Phoenix, Arizona, pp. 143-147, October 1988.

[46] Biggerstaff, T.J., Mitbander, B.G. and Webster, D.E., *Program Understanding and the Concept Assignment Problem*, Communications of the ACM, Vol. 37, No. 5, pp. 72-83, May 1994.

[47] Bigus, J.P, *Data Mining with Neural Networks*, McGraw-Hill, 1996.

[48] Boehm, B.W., *Software Engineering*, IEEE Transactions Computers, pp. 1226-1241, December, 1976.

[49] Boehm, B.W., *The Economics of Software Maintenance.* In: Ed. R.S. Arnold, Proceedings, Workshop on Software Maintenance, pp. 9-37, IEEE Computer Society Press, 1983.

[50] Boehm, B.W., *A Spiral Model of Software Development and Enhancement*, Computer, May 1988.

[51] Boldyreff, C. and Zhang, Z., *From Recursion Extraction to Automated Commenting*, Chapter 12 in Software Reuse and Revere Engineering in Practice, pp. 253-270, Ed. P.A.V. Hall, Chapman & Hall, 1992.

[52] Boldyreff, C., *The AMES Approach to Application Understanding - a Case Study*, pp. 182-191, in Proceedings International Conference on Software Maintenance, Nice, France, 1995.

[53] Boldyreff, C., Burd, E.L., Hather, R.M., Munro, M. and Younger, E., *Greater Understanding Through Maintainer Driven Traceability*, In Proceedings 4th Workshop on Program Comprehension, pp. 100-106, Berlin, Germany, March 1996.

[54] Booch, G., *Leaving Kansas*, IEEE Software, pp. 32-35, January-February, 1998.

[55] Brereton, P. and Singleton, P., *Deductive Software Building*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[56] Brereton, O.P., Budgen, D. and Hamilton, G.W., *Hypertext: the Next Maintenance Mountain?* In Proceedings 2000 and Beyond: 10th Annual European Software Maintenance Workshop, University of Durham, 1996.

[57] British Standards Institution, *British Standard Code of Practice for Configuration Management of Computer-Based Systems*, BS 6488, 1984.

[58] British Telecom Research Laboratories, *DOCMAN Intermediate File Format Specification*, Martlesham Heath, 1989.

[59] Brooks, F.P., *The Mythical Man Month: Essays on Software Engineering*, Reading, Mass., Addison-Wesley, 1982.

[60] Brooks, R., *Toward a Theory of the Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, pp. 543-554, 1983.

[61] Brooks, F.P., *No Silver Bullet - Essence and Accidents of Software Engineering*, IEEE Computer, Vol. 20, No. 4, pp. 10-20, April 1987.

[62] Brown, A.W., *CASE in the 21st Century: Challenges Facing Existing CASE Vendors*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 268-278, London, July 1997.

[63] Brown, P., *Integrated Hypertext and Program Understanding Tools*, IBM Systems Journal, Vol. 30, No. 3, pp. 363-392, 1991.

[64] Buckle, J.K., *Software Configuration Management*, Macmillan, London, 1982.

[65] Buckley, F.J., *Implementing Configuration Management: Hardware, Software and Firmware*, IEEE Press, 1993.

[66] Buffenbarger, J., *Syntactic Software Merging*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995

[67] Buschman, F., *What is a Pattern,* Object Expert, Vol. 1, No. 3, pp. 17-18, March-April, 1996.

[68] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P and Stal, M., *Pattern-Oriented Software Architecture*, John Wiley, 1996.

[69] Buschman, F., *Design Patterns: Finding and Using Patterns,* Object Expert, Vol. 1, No. 6, pp. 46-47, 61, September-October, 1996.

[70] Byrne, E.J., *A Conceptual Foundation for Software Re-engineering*, In Proceedings Conference on Software Maintenance, pp. 226-235, Orlando, Florida, 1992.

[71] Cabletron, *Network Management Based on Artificial Intelligence*, Section 12 in The Cabletron Systems Guide to Local Area Networking, Cabletron Systems Limited, 1992

[72] Cagan, M., *Untangling Software Configuration Management*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[73] Calow, H., *Managing the Software Asset: An Approach to Support and Maintenance*, In notes: Fourth Software Maintenance Workshop, Eds. S. Cooper, R. Freeman, R.. Kenning, D. Hindley and M. Munro, Centre for Software Maintenance, University of Durham, September 1990.

[74] Canfora, G. and Cimitile, A., *Algorithms for Program Dependence Graph Production*, In Proceedings International Conference on Software Maintenance, pp. 157-166, Nice, France, IEEE Computer Society Press, 1995.

[75] Canfora, G., Mancini, L. and Tortorella, M., *A Workbench for Program Comprehension during Software Maintenance*, In Proceedings 4th Workshop on Program Comprehension, pp. 30-39, Berlin, Germany, March 1996.

[76] Capretz, M.A.M. and Munro, M., *COMFORM - A Software Maintenance Method Based on the Software Configuration Management Discipline*, pp. 183-192, In Proceedings Conference on Software Maintenence, Florida, November, 1992.

[77] Card, D.N., *Learning from our Mistakes with Defect Causal Analysis*, IEEE Software, pp. 56-63, January-February, 1998.

[78] Catell, R.G.G., *Object Data Management: Object Oriented and Extended Relational Database Systems*, Addison-Wesley, 1991.

[79] Chan, A.K.F. and Hung, S-L., *Software Configuration Management Tools*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 238-250, London, July 1997.

[80] Chapin, N., *Changes in Change Control*, in Proceedings International Conference on Software Maintenance, pp. 246-253, Miami, Florida, October, 1989.

[81] Chase, R.P., Jr., *System Modelling*, Summary of Plenary Discussion International Workshop on Software Version and Configuration Control, Grassau, pp. 172-174, January 1988.

[82] Chaudhri, A.B. and Osmon, P., *Databases for a New Generation,* Object Expert, Vol. 1, No. 3, pp. 26-31, March-April, 1996.

[83] Cheng, K. and Rowe, W.B., *Hypermedia as a Tool*, Computing & Control Engineering Journal, Vol. 6, No. 3, pp. 123-130, June 1995.

[84] Chikofsky, E.J. and Cross II, J.H., *Reverse Engineering and Design Recovery: a Taxonomy*, IEEE Software, Vol. 7, No. 1, pp. 13-17, January 1990.

[85] Chikofsky, E.J., *Software Engineering Programs for the Next Century*, Computer, Vol. 29, No. 10, pp. 117-118, October, 1996.

[86] Chweh, C., *Generations Ahead: Cooperative Information Systems*, IEEE Expert, pp. 82-83, September-October, 1997.

[87] Cimitile, A., deLucia, A. and Munro, M., *Identifying Reusable Functions using Specification Driven Program Slicing*, in Proceedings Conference on Software Maintenance, pp. 124-133, 1995.

[88] Citrin, W., Santiago, C. and Zom, B., *Scalable Interfaces to Support Program Comprehension*, In Proceedings 4th Workshop on Program Comprehension, pp. 123-132, Berlin, Germany, March 1996.

[89] Cleal, D., *Optimising Relational Database Access*, Object Expert, Vol. 1, No. 3, pp. 33-38, March-April, 1996.

[90] Clemm, G.M., ODIN - *An Extensible Software Environment: Report and Users Reference Manual*, Technical Report CU-CS-262-84, Department of Computer Science, University of Colorado, March 1984.

[91] Clemm, G.M., *KEYSTONE: A Federated Software Environment*, In Proceedings Workshop on Software Environments for Programming-in-the-Large, pp. 80-88, Massachusetts, June 1985.

[92] Clemm, G.M., *The Odin Specification Language*, International Workshop on Software Version and Configuration Control, Grassau, pp. 144-158, January 1988.

[93] Clemm, G. and Osterweil, L., *A Mechanism for Environment Integration*, IEEE Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 1-25, January, 1990.

[94] Clemm, G.M., *The Odin System*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[95] Cohen, E.S., Soni, D.A., Gluecker, R., Hasling, W.M., Schwanke, R.W., Wagner, M.E., *Version Management in Gypsy*, pp. 201-215, 1988 ACM.

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

[97] Colter, M. *Keynote Speech - The Business of Software Maintenance*, In Notes: Second Software Maintenance Workshop, Centre for Software Maintenance, University of Durham, Eds. Callis, F., Cooper, S. Kenning, R. and Munro, M., September 1988

[98] Computer Science Technology Board, *Scaling Up: A Research Agenda for Software Engineering*, Communications of the ACM, Vol. 33, No. 3, pp. 281-293, March 1990.

[99] Conklin, J., *Hypertext: An Introduction and Survey*, IEEE Computer Vol. 20, No. 9, September 1987, pp. 17-41.

[100] Conradi, R. *CM for Distributed and Heterogeneous Systems*, Summary of Plenary Discussion International Workshop on Software Version and Configuration Control, pp. 310-312, Grassau, January 1988.

[101] Conradi, R. Didriksen, T.M., Karlsson, E-A., Lie, A., Westby, P.H. *Design of the Kernel EPOS Software Engineering Environment.*

[102] Cookman, D.R., *A Lifespan Experience*, Proceedings Software Tools'89 Conference, Wembley, London, pp.119-125, 1989.

[103] Cooper, R., *The Fundamentals of Configuration Management in a Software Development Environment*, Proceedings Software Tools'89 Conference, Wembley, London, pp.93-101, 1989.

[104] Cooper, S.D. and Munro, M., *Software Change Information for Maintenance Management*, in Proceedings International Conference on Software Maintenance, pp. 279-287, Miami, Florida, October, 1989.

[105] Cooprider, L.W., *The Representation of Families of Software Systems*, Ph.D. Thesis, Carnegie-Mellon University, Computer Science Department, April 1979.

[106] Copeland, G. and Maier, D., *Making Smalltalk a Database System*, ACM SIGMOD International Conference, June 1984.

[107] Coplien, J.O., *Idioms and Patterns as Architectural Literature*, IEEE Software, pp.36-42, January 1997.

[108] Corbi, T.A., *Program Understanding: Challenge for the 1990s*, IBM Systems Journal, Vol. 28, No. 2, pp. 294-306, 1989.

[109] Cornelius, B.J., Munro, M. and Robson, D., *An Approach to Software Maintenance Education*, Software Engineering Journal, Vol. 4, No. 4, pp. 233-40, July 1988.

[110] Cort, G., *The Los Alamos Hybrid Environment: An Integrated Development / Configuration Management System*, pp. 10-17, IEEE 1985.

[111] Cox, B., *Superdistribution and the Economics of Bits*, IEEE Software, pp. 22-24, January, 1997.

[112] Cox, E., *The Fuzzy Systems Handbook: A Practitioners Guide to Building, Using and Maintaining Fuzzy Systems*, Academic Press, 1994.

[113] Crosby, P., *Quality is Free*, McGraw-Hill, 1979.

[114] Crowder, R.M, Hall, W., Heath, I., Bernard, R. and Gaskell, D., *A Hypermedia Maintenance Information System*, Computing & Control Engineering Journal, pp. 121-128, June 1996.

[115] Curtis, B., *Maintaining the Software Process*, In Proceedings Conference on Software Maintenance, pp. 2-8, Orlando, Florida, November, 1992.

[116] Cussons, J., *Machine Learning*, Computing and Control Engineering Journal, Vol. 7, No. 4., pp. 164-168, August 1996.

[117] deBaud, J-M. and Rugaber, S., *A Software Re-Engineering Method using Domain Models*, In Proceedings Conference on Software Maintenance, pp 204-213, Nice, France, 1995.

[118] DeJesus, E., *Solving for the Year 2000*, Byte, p. 75, January 1998.

[119] Dekleva, S., *Delphi Study of Software Maintenance Problems*, In Proceedings Conference on Software Maintenance, pp. 10-17, Orlando, Florida, November, 1992.

[120] Dellarocas, C., *The SYNTHESIS Environment for Component-Based Software Development*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 434-443, London, July 1997.

[121] Delobel, C., Lecluse, C. and Richard, P., *Databases: from Relational to Object-Oriented Systems*, International Thompson Publishing, 1995.

[122] deLucia, A., Fasolino, A.R. and Munro, M., *Understanding Function Behaviours through Program Slicing*, in Proceedings 4th Workshop on Program Comprehension, pp. 9-18, Berlin Germany, March 1996.

[123] deMarco, T., *The Role of Software Development Methodologies: Past, Present and Future*, pp. 2-4, in Proceedings 18th International Conference on Software Engineering, Berlin, Germany, IEEE Computer Society Press, March 1996.

[124] Deming, W.E., *Out of Crisis*, MIT Centre for Advanced Engineering Study, 1982.

[125] Dempsey, K-A., McCrindle, R.J. and Williams, S., *Multimedia multi-Platform, Identification and Tracking System (Mu^{2}PITS)*, Final Project Report, Supervised by R.J. McCrindle, Department of Computer Science, the University of Reading, 1996.

[126] DeRemer, F. and Kron, H.H., *Programming-in-the-Large versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. SE-2, pp. 80-86, June 1976.

[127] Dittrich, K., and Lockmann, P., *Damokles - a Database System for Software Engineering Environments*, in Proceedings IFIP WG2.4 International Workshop on Advanced Programming Environments, Trodheim Norway, June 1986.

[128] Doggett, S., *Investigation into Multimedia Maintenance Interfaces to Assist Program Comprehension*, Final Project Report, University of Reading, Supervised by R.J.McCrindle, 1996

[129] Dunlop, A., Papiani, M. and Hey, T., *Providing Access to a Multimedia Archive Using the World Wide Web and an Object-Relational Database Management System*, Computing and Control Engineering Journal, Vol. 7., No. 5., pp.221-226, October 1996.

[130] Economou, G.P.K., Goumas, P.D. and Spiropoulos, K., *A Novel Medical Decision Support System*, Computing & Control Engineering Journal, pp. 177-183, August 1996.

[131] Eick, S.G. and Ward, A., *An Interactive Visualisation for Message Sequence Charts,* In Proceedings 4th Workshop on Program Comprehension, pp. 2-8, Berlin, Germany, March 1996.

[132] Ellis, R., *Data Abstraction and Program Design: from Object-Based to Object-Oriented Programming*, 2nd Edition, UCL Press, 1996.

[133] Erickson, V.B., *Build - a Software Construction Tool*, AT&T Bell Laboratories Technical Journal, Vol. 63, No. 6, July/August 1984.

[134] Estublier, J. and Belkahtir, N., *Experience with a Database of Programs*, In Proceedings of the Software Engineering Symposium on Practical Software Development Environments. ACM SIGPLAN Notices, Vol. 22, No. 1, pp 84-91, January 1984.

[135] Estublier, J. *A Configuration Manager: The Adele Database of Programs*, In Proceedings of the GTE Workshop on Programming Environments for Programming-in-the-Large, Hariwchport, June 1985.

[136] Estublier, J., *Configuration Management - The Notion and the Tools*, International Workshop on Software Version and Configuration Control, Grassau, pp. 38-61, January 1988.

[137] Estublier, J. and Casallas, R., *The Adele Configuration Manager*, Chapter 4, pp. 99-139, Trends in Software, John Wiley, 1994.

[138] Estublier, J., *Preface - Software Configuration Management Workshop*, ICSE SCM-4 and SCM-5 Workshops, Lecture Notes in Computer Science, Springer-Verlag, 1995.

[139] Estublier, J., *Process Session*, ICSE SCM-4 and SCM-5 Workshops, Lecture Notes in Computer Science, Springer-Verlag, 1995.

[140] Faden, M., *Tools for Managing Change That Don't Mess up the System*, DEC User, pp. 44-47, November 1987.

[141] Faltenbacher, W., *Aspects of Computer-Aided Configuration Management with KMS*, International Workshop on Software Version and Configuration Control, Grassau, pp. 369-380, January 1988.

[142] Federal Information Processing Standards, *Guideline on Software Maintenance*, FIPS Pub 106, U.S. Department of Commerce, June 1984.

[143] Feiler, P.H., *Life Cycle Oriented CM*, Summary of Plenary Discussion International Workshop on Software Version and Configuration Control, Grassau, pp. 305-309, January 1988.

[144] Feiler, P.H., Smeaton, R., *Managing Development of Very Large Systems: Implications on Integrated Environments*, International Workshop on Software Version and Configuration Control, Grassau, pp. 62-82, January 1988.

[145] Feldman, S.I., *Evolution of Make*, International Workshop on Software Version and Configuration Control, Grassau, pp. 412-416, January 1988.

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

[147] Fickas, S., *Knowledge-Based Version Control: an Automated Design Perspective*, International Workshop on Software Version and Configuration Control, Grassau, pp. 461-465, January 1988.

[148] Fletton, N.T. and Munro, M., *Redocumenting Software Systems Using Hypertext Technology*, In Proceedings Conference on Software Maintenance, pp. 54-59, Phoenix, Arizona, IEEE Computer Society Press, 1988.

[149] Foster, J.R. and Munro, M., *A Documentation Method Based on Cross-Referencing*, In Proceedings Conference on Software Maintenance, pp. 181-185, Texas, 1987, IEEE Computer Society Press.

[150] Foster, J. R., *A 7-Level Model for Software Maintenance*, In Notes 3rd, Software Maintenance Workshop, Eds. Callis, F., Cooper, S, Freeman, R, Kenning, R., Munro, M., Durham 1989.

[151] Foster, J.R., Jolly, A.E.P. and Norris, M.T., *An Overview of Software Maintenance,* British Telecom Technology Journal, Vol. 7, No. 4, pp. 37-46, October 1989

[152] Fowler, G., *The Fourth Generation Make*, In Proceedings Summer USENIX Conference, pp. 159-174, 1985.

[153] Fowler, G., *A Case for Make*, Software Practice and Experience, Vol. 20, No. 1, pp. 35-46, June 1990.

[154] Freeman, P.A., *Conceptual Analysis of the Draco Approach to Constructing Software Systems*, IEEE Transactions on Software Engineering, 1987.

[155] Fu, L., *Integration of Knowledge and Neural Heuristics*, AI Magazine, pp. 85-88, Winter 1996.

[156] Fuggetta, A., Bandinelli, S. and Nitto, E.D., *Policies and Mechanisms to Support Process Evolution in PSEE*, in Proceedings 3rd International Conference on the Software Process, IEEE Computer Society Press, 1994.

[157] Fuller, A.R., *Groupware in the Making*, in: Computer Support for Co-operative Work, eds., Spurr, K., Layzell, P., Jennison, L. and Richards, N., pp.71-88, Wiley, 1995.

[158] Gadonna, C., *Mistral User Manual v1*, LGI, ESPRIT Project 53277-REBOOT, May 1995.

[159] Garg. P.K. and Scacchi, W., *A Software Hypertext Environment for Configured Software Descriptions*, International Workshop on Software Version and Configuration Control, Grassau, pp. 326-343, January 1988.

[160] Garg, P.K. and Scacchi, W., *ISHYS Designing an Intelligent Software Hypertext System*, IEEE Expert, pp. 52-63, Fall, 1989.

[161] Garlan, D., *Introduction*, in: Summary of the Dagstuhl Workshop on Software Architecture, Software Enginering Notes, Vol. 20, No. 3, pp. 63-83, July 1995.

[162] Garlan, D., Tichy, W. and Paulish, F., *Summary of the Dagstuhl Workshop on Software Architecture*, Software Enginering Notes, Vol. 20, No. 3, pp. 63-83, July 1995.

[163] Georges, M., *Message from the General Chair*, Proceedings, International Conference on Software Maintenance, France, 1995, IEEE Computer Society Press, 1995.

[164] Geschke, C.M, Morris, J.H., Satterwaite, E.H., *Early Experiences with MESA*, Communications of the ACM, pp. 540-552, August 1987.

[165] Ghezzi, C., Jazayayeri, M. and Mandrioli, D., *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[166] Gibbs, S. and Breitender, C., *Large, Multimedia Programming - Concepts and Challenges*, in Proceedings IEEE Conference on Software Engineering, pp. 439-440, Berlin, Germany, March 1996.

[167] Gibbs, W.W., *Software's Chronic Crisis*, Scientific American, Vol. 271, No. 3, pp. 72-81, September 1994.

[168] Glass, R.L., *Position Paper: Software Maintenance is a Solution, Not a Problem*, in Proceedings International Conference on Software Maintenance, pp. 224-225, Miami, Florida, October, 1989.

[169] Glass, R.L., *In Praise of Practice*, IEEE Software, pp. 30-31, January-February, 1998.

[170] Godfrey, K., *The Ever-Changing Face of Integrated Microprocessors - an Evolutionary Story*, Computing & Control Journal, pp. 153-160, June 1996.

[171] Green, P., Morris, D. and Evans, G., *Software Technology for Embedded Systems*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 402-410, London, July 1997.

[172] Grey, J., *Evolution of Data Management*, Computer, Vol. 29, No. 10, pp, 38-46, October 1996.

[173] Grimes, J. and Potel, M., *Software is Headed Towards Object-Oriented Components*, IEEE Computer, pp. 24-25, August 1995.

[174] Griswold, W. G., Atkinson, D.C. and McCurdy, C., *Fast, Flexible, Syntactic Pattern Matching and Processing*, In Proceedings 4th Workshop on Program Comprehension, pp. 144-153, Berlin, Germany, March 1996.

[175] Grosky, W.I., *Managing Multimedia Information in Database Systems*, Communications of the ACM, Vol. 40, No. 12, pp. 73-80, December 1997.

[176] Grudin, J., *Computer-Supported Cooperative Work*, Computer, pp. 19-26, May 1994.

[177] Gulla, B., *Improved Maintenance Support by Multi-Version Visualisations*, In Proceedings Conference on Software Maintenance, pp. 376-383, Florida, 1992.

[178] Habermann, A.N. and Notkin, D., *Gandalf: Software Development Environment*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 12, pp. 1117-1127, December 1986.

[179] Hall, P.A.V., *Software Reuse, Reverse Engineering and Re-engineering*, Chapter 1 in Software Reuse and Reverse Engineering in Practice, pp. 3-31, Ed. P.A.V. Hall, Chapman & Hall, 1992.

[180] Hall, P. and Weedon, R., *Object Oriented Module Interconnection Languages*, In Proceedings 2nd International Workshop on Software Reuse, Lucca, Italy, pp. 29-38, IEEE Press, 1993.

[181] Hall, W. and Davis, H., *Hypermedia Link Services and their Application to Multimedia Information Management*, Information and Software Technology, Vol. 36, No. 4., pp. 197-202, 1994.

[182] Halladay, S. and Wiebel, M., *Object-Oriented Software Engineering*, R & D Publications, 1993.

[183] Harjani, D-R. and Queille, J-P, *A Process Model for the Maintenance of Large Space Systems Software*, In Proceedings Conference on Software Maintenance, pp. 127-136, Orlando, Florida, November, 1992.

[184] Harrison, W., Gens, C. and Gifford, B., *pRETS: a Parallel Reverse Engineering Toolset for Fortran*, Journal of Software Maintenance: Research and Practice, Vol. 5, No. 1, pp 37-57, March 1993.

[185] Hart, J.M. and Pizzarello, A., *A Scaleable, Automated Process for Year 2000 System Correction*, In Proceedings International Conference on Software Engineering, pp. 475-484, Berlin, Germany, 1996.

[186] Hart, P.E. and Graham, J., *Query-Free Information Retrieval*, IEEE Expert, pp. 32-37, September 1997.

[187] Hatton, L., *Software Failures - Follies and Fallacies*, IEE Review, pp. 49-52, March 1997.

[188] Hayes-Roth, B., Pfflеger, K., Lalanda, P., Morignot, P. and Balabnovic, M., *A Domain-Specific Software Architecture for Adaptive Intelligent Systems*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 288-301, April 1995.

[189] Haziza, M, Voidrot, J.F., Minor, E., Pofelski, L. and Blazy, S., *Software Maintenance: An Analysis of Industrial Needs and Constraints*, In Proceedings Conference on Software Maintenance, pp. 18-25, Orlando, Florida, November, 1992.

[190] Heimbigner, D., *A Graph Transform Model for Configuration Management Environments*, pp. 216-225, ACM 1988.

[191] Hekmatpour, S. and Ince, D., *Software Prototyping, Formal Methods and VDM*, Addison Wesley, 1988.

[192] Henderson-Sellers, B., *A Book of Object-Oriented Knowledge: An Introduction to Object-Oriented Software Engineering*, 2nd Edition, Prentice-Hall, 1997.

[193] Hendler, J.A., *Intelligent Agents: Where AI Meets Information Technology*, IEEE Expert, pp. 20-23, December 1996.

[194] Herbsleb, J.D. and Goldenson, D.R., *A Systematic Survey of CMM Experience & Results*, in Proceedings 18th International Conference on Software Engineering pp. 323-330, Berlin, Germany, March, 1996.

[195] Hoare, C.A.R., *An Axiomatic Approach to Computer Programming*, CACM, Vol. 12, No. 10, pp. 567-580, October 1969.

[196] Howden, W.E., and Pak, S, *Problem Domain, Structural and Logical Abstractions in Reverse Engineering*, in Proceedings Conference on Software Maintenance, pp. 214-224, Orlando, Florida, 1992.

[197] Hume, A., *Mk: a Sucessor to Make*, In Proceedings Summer USENIX Conference, Phoenix, Arizona, June 1987.

[198] IEEE *Standard 1219-1993 - Standard for Software Maintenance*, IEEE, 1993.

[199] IEEE, *Draft Standard for Software Maintenance*, IEEE, New York.

[200] Iivari, J, *Why are CASE Tools not Used?*, Communications of the ACM, Vol. 39, No.10, pp. 94-103, October 1996.

[201] Ino, M., *Current State of Software Maintenance in Japan: In Depth View*, In Proceedings Conference on Software Maintenance, pp. 27-29, Florida, November, 1992.

[202] Isakowitz, T. and Kauffman, R.J., *Supporting Search for Reusuable Software Objects*, IEEE Transactions on Software Engineering, Vol. 22, No. 6, pp. 407-423, June 1996.

[203] Jackson, M., *Will there ever be Software Engineering*, IEEE Software, pp. 36-39, January-February, 1998.1998.

[204] Jain, R., *Visual Information Management*, Communications of the ACM, Vol. 40, No. 12, pp. 31-32, December 1997.

[205] Jameson, K., *Multi-Platform Code Management*, Release 1.0, O'Reilly & Associates, 1994.

[206] Jennings, N.R. and Wooldridge, M., *Software Agents*, pp. 17-20, IEE Review, January 1996.

[207] Jesty, P.H. and Hobley, K.M., *System Architecture and its use in Safety-Related Telematics Systems*, Computing and Control Engineering Journal, Vol. 9, No. 1, pp. 4-7, February 1998.

[208] Johnson, B., Ornburn, S. and Rugaber, S., *A Quick Tools Strategy for Program Analysis and Software Maintenance*, In Proceedings Conference on Software Maintenance, pp. 206-213, Florida, November, 1992.

[209] Jones, C., *Programming Productivity*, McGraw Hill, 1986.

[210] Juran, J.M., *Juran on Leadership for Quality*, Free Press, 1989.

[211] Kador, J., *Change Control and Configuration Management*, Systems Development, May 1989.

[212] Kaiser, G.E. and Feiler, P.H., *An Architecture for Intelligent Assistance in Software Development*, In Proceedings 9th International Conference on Software Engineering, Monterey, California, pp. 180-188, 1987.

[213] Kaiser, G.E., Feiler, P.H., Popovich, S.S., *Intelligent Assistance for Software Development and Maintenance*, IEEE Software, pp. 40-49, May 1988.

[214] Karakostas, V., *The Use of Application Domain Knowledge for Effective Software Maintenance*, in Proceedings Conference on Software Maintenance, pp.170-176, San Diego, California, November 1990.

[215] Kaye, J., *Neural Networks: the Thinking Computer's Crumpet*, Computer Weekly, pp. 36, 23rd January 1997

[216] Kaynak, O. and Rudas, I., *Soft Computing Methodologies and their Fusion in Mechatronic Products*, Computing & Control Engineering Journal, Vol. 6, No. 2., pp 68-72, April 1995.

[217] Kerth, N.L. and Cunningham, W., *Using Patterns to Improve Our Architectural Vision*, IEEE Software, pp. 53-60, January 1997.

[218] Khoshgoftaar, T.M, Szabo, R.M. and Guasti, P.J., *Exploring the Behaviour of Neural Network Software Quality Models*, Software Engineering Journal, Vol. 10, No. 3, pp.89-96, May 1995.

[219] Kim, W., Banerjee, J, Chou, H., Garza., J. and Woelk, D., *Composite Object Support in an Object-Oriented Database System*, ACM SIGMOD International Conference, May 1988.

[220] Kirby, M., *Prospective Kernel - an Open-Ended Project Support Environment*, International Workshop on Software Version and Configuration Control, Grassau, pp. 391-400, January 1988.

[221] Kitchenham, B., Pickard, L. and Pfleeger, S.L., *Case Studies for Method and Tool Evaluation*, IEEE Software, pp. 53-62, July 1995.

[222] Kitchenham, B., Linkman, S. and Law, D., *DESMET: a Methodology for Evaluating Software Engineering Methods and Tools*, Computing & Control Engineering Journal, pp. 120-128, June 1997.

[223] Kliene, S.C. and Keller, W.A., *Problem Report Forms: A System for Software Configuration Environments*, ACM Sigsoft Software Engineering Notes, Vol. 13, No. 4, pp. 79-84, October 1988.

[224] Krane, S., *Towards a Graph Transform Model for Configuration Management Environments*, International Workshop on Software Version and Configuration Control, Grassau, pp. 438-441, January 1988.

[225] Kraut, R.E. and Streeter, L.A., *Co-ordination in Software Development*, Communications of the ACM, Vol. 38, No. 3, pp. 69-81, Match 1995.

[226] Krulwich, B. and Burkey, C., *The InfoFinder Agent: Learning User Interests through Heuristic Phrase Extraction*, IEEE Expert, pp.22-27, September-October, 1997.

[227] Kuvaja, P. and Koch, G, *Maturity of Maintenance*, in Proceedings Conference on Software Maintenance, pp. 259-260, Orlando, Florida, 1992.

[228] Lano, K. and Haughton, H., *Reverse Engineering and Software Maintenance, a Practical Approach,* McGraw Hill, 1994.

[229] Lano, K. and Malic, N., *Reengineering Legacy Applications using Design Patterns*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 326-338, London, July 1997.

[230] Lanubile, F. and Visaggio, G., *Iterative Reengineering to Compensate for Quick-fix Maintenance*, in Proceedings International Conference on Software Maintenance, pp. 140-146, Nice, France, 1995.

[231] Lauer, H.C. and Satterthwaite, *The Impact of MESA on System Design*, In Proceedings of the 4th International Conference on Software Engineering, pp. 174-182, Munich, Germany, September 1979.

[232] Laufmann, S.C., *Towards Agent-Based Software Engineering for Information-Dependent Enterprise Applications*, IEE Proceedings Software Engineering, Vol. 144, No. 1, pp. 38-50, February 1977.

[233] Layzell, P.J. and Macaulay, L.A., *An Investigation into Software Maintenance Perception and Practices*, in Proceeding Conference on Software Maintenance, pp. 130-140, San Diego, California, 1990.

[234] Layzell, P.J. and Macaulay, L.A., *An Investigation into Software Maintenance - Perceptions and Practices*, Journal of Software Maintenance: Research and Practice, Vol. 6., No. 3, pp. 105-120, 1994.

[235] Leblang, D.B., *Computer Aided Software Engineering in a Distributed Workstation Environment*, In Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Notices, Vol. 19, No. 5, pp. 104-112, May 1984.

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

[237] Leblang, D.B. and Levine, P.H., *Software Configuration Management: Why it is Needed and What Should it Do?*, Software Configuration Management, ICSE SCM-4 and SCM-5 Selected Papers, Lecture Notes in Computer Science, ED. J. Estublier, Springer-Verlag, 1995.

[238] Legent Corporation, *Endevor WSX Product Overview*, www@http: //www.sv.legent.com/Info/Ewsx/Ewsx.html.

[239] Lehman, M.M. and Belady, L., *Program Evolution, Process of Software Change,* Academic Press, 1985.

[240] Lehman, M.M., *On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle*, The Journal of Systems and Software, Vol. 1, pp. 213-221, 1980.

[241] Lehman, M.M., *Software'e Future: Managing Evolution*, IEEE Software, pp. 40-44, January-February, 1998.

[242] Lewis, T.G., *Where is Computing Heading?*, IEEE Computer, pp. 59-63, 1994.

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

[244] Littlewood, B. and Strigini, L., *The Risks of Software*, Scientific American Special Issue : The Computer in the 21st Century, pp. 180-185, 1995.

[245] Littman, D.C., Pinto, J., Levovsky, S. and Solway, E., *Mental Models and Software Maintenance.* In Empirical Studies of Programmers, Eds. E. Soloway and S. Iyengar, Ablex, Norwood, pp. 80-96, 1986.

[246] Lockwood, R., *The Groupware Market*, in: Computer Support for Co-operative Work, eds., Spurr, K., Layzell, P., Jennison, L. and Richards, N., pp.3-18, Wiley, 1995.

[247] Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera J., Bryan, D. and Mann, W., *Specification and Analysis of System Architecture using Rapide*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 336-355, April 1995.

[248] Lynch, L. and Skelton, S., *Teleworking in the Information Society*, Computing and Control Engineering Journal, Vol. 7, No. 1, pp. 33-38, February 1996.

[249] MacKay, S.A., *The State of the Art in Concurrent Distributed Configuration Management*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995

[250] Madar, T.B., *Pro-CM, An Ipse based Configuration Management System*, M.Phil, Department of Computer Science, University College of Wales, Aberystwyth, 1988.

[251] Maes, P., *Intelligent Software*, Scientific American, Special Issues on Key Technologies for the 21st Century, pp. 66-68, September 1995.

[252] Maher, M.L. and de Silva Garza, A.G., *Case-Based Reasoning in Design*, IEEE Expert, pp. 34-40, March-April 1997.

[253] Mahler, A. and Lampen, A., *An Integrated Toolset for Engineering Software Configurations*, pp. 191-200, ACM 1988.

[254] Mahler, A. and Lampen, A., *Shape - a Software Configuration Management Tool*, 1988.

[255] Mansfield, R., *Mastering Word 97*, 4th Edition, Sybex, 1997.

[256] Manns, T. and Coleman, M., *Software Quality Assurance*, 2nd Edition, MacMillan, 1996.

[257] Martin, R.J. and Osborne, W.M., *Guidance on Software Maintenance*, National Bureau of Standards Special Publication 500-106, U.S. Department of Commerce, 1983.

[258] Marzullo, K., Wiebe, D., *Jasmine: a Software Modelling System*, In Proceedings Software Engineering Symposium on Practical Software Development Environments, December 1986. SIGPLAN Notices, pp. 121-130, January 1987.

[259] McCabe, T., *Cyclomatic Complexity and the Year 2000*, IEEE Software, pp. 115-119, May 1996.

[260] McCarthy, R., *Applying the Techniques of Configuration Management to Software*, Vol. 11, Part 4, pp. 23-28, October 1975.

[261] McConnell, S., *The Art, Science, and Engineering of Software Development*, IEEE Software, pp. 118-120, January-February, 1998.

[262] McCrindle, R.J. (Kenning) and Munro, M., *Configuration Management - The State of the Art: Part I - Management Aspects of Software Configuration Management*, Technical Report, Centre for Software Maintenance, University of Durham, 1988.

[263] McCrindle, R.J. (Kenning) and Munro, M., *Configuration Management - The State of the Art: Part II - Technical Aspects of Software Configuration Management*, Technical Report, Centre for Software Maintenance, University of Durham, 1989.

[264] McCrindle, R.J. (Kenning) and Munro, M., *Towards Configuring Operational Systems*, In Proceedings Conference on Software Tools 1990, Wembley, London, December 1990.

[265] McCrindle, R.J. (nee Kenning) and Munro, M., *Understanding the Configurations of Operational Systems*, In Proceedings Conference on Software Maintenance, pp. 20-27, San Diego, California, November 1990.

[266] McCrindle, R.J. (Kenning) and Munro, M., *PISCES - An Inverse Configuration Management System*, Chapter 17 in Reuse and Reverse Engineering In Practice, Ed. P.A.V. Hall, Chapman & Hall, 1992.

[267] McCrindle, R.J., and Doggett, S., *Investigation into Multimedia Maintenance Interfaces to Assist Program Comprehension*, In Proceedings 1st program Comprehension Workshop, Durham, June 1995.

[268] McCrindle, R.J., and Doggett, S. *The Multimedia Maintenance Interface System*, Presented at BCS Seminar on Software Creativity (and to appear in forthcoming book on Software Creativity), May 1996.

[269] McCrindle, R.J., and Doggett, S., *The Reality of the Virtual Maintainer*, In Proceedings 10th European Workshop on Software Maintenance, Durham, September 1996.

[270] McCrindle, R.J, *The M^4 System*, Paper being authored, 1998.

[271] McCrindle, R.J., and Munro, M. *Inverse Software Configuration Management*, Paper being authored, 1998.

[272] Mellor, P., *A Model of the Problem or a Problem with the Model*, Computing and Control Engineering Journal, Vol. 9, No. 1, pp. 8-18, February 1998.

[273] Mellor, S.J. and Johnson, R., *Why Explore Object Methods, Patterns, and Architectures?*, IEEE Software, pp. 27-30, January 1997.

[274] Microsoft Corporation, *Manuals for Microsoft Windows Software Development Kit Version 3.0*, 1990.

[275] Middleditch, M., *Flexibility Breeds Success*, Systems International, pp.43-44, May 1989.

[276] Minor, E., *The Object-Oriented Paradigm for Software Maintenance: No Silver Bullet, but a Good Investment*, In Proceedings Conference on Software Maintenance, pp. 81-82, Florida, IEEE Computer Society Press, 1992.

[277] Minsky, N., *Configuration Management by Consensus: an Application of Law-Governed Systems*, NSF CCR-8807803 Report, July 1989.

[278] Monk, S.R. and Sommerville, I., *A Model for Versioning of Classes in Object-Oriented Databases*, In Lecture Notes in Computer Science, Vol. 618, pp.42-58, 1992.

[279] Monroe, R.T., Kompanek, A., Melton, R. and Garlan, D., *Architectural Styles, Design Patterns, and Objects*, IEEE Software, pp. 43-52, January 1997.

[280] Montes, J. and Haque, T., *A Configuration Management System and More!,* International Workshop on Software Version and Configuration Control, Grassau, pp. 217-227, January 1988.

[281] Moreira, A.M.D. and Clark, R.G., *Adding Rigour to Object-Oriented Analysis*, Software Engineering Journal, pp. 270-280, September 1996.

[282] Moriconi, M., Quon, R. and Riemenschneider, R.A., *Correct Architecture Refinement,* IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 356-xxx, April, 1995

[283] Munro, M., *Software Maintenance, Reuse and Reverse Engineering*, Chapter 30 in Software Reuse and Reverse Engineering in Practice, pp. 573-584, Ed. P.A.V. Hall, Chapman & Hall, 1992.

[284] Murray, J.T. and Murray, M.J., *The Year 2000 Computing Crisis*, McGraw Hill, 1996.

[285] Nado, R., Chams, M., Delisio, J. and Hamscher, W., *COMET: An Application of Model-Based Reasoning to Accounting Systems*, AI Magazine, pp. 55-63, Winter 1996.

[286] Narayanaswamy, K. and Scacchi, W., *Maintaining Configurations of Evolving Software Systems*, IEEE Transactions on Software Engineering, Vol SE-13, No. 3, pp. 324-334, March 1987.

[287] Narayanaswamy, K., *Version Control in the Common Lisp Framework*, International Workshop on Software Version and Configuration Control, Grassau, pp. 83-97, January 1988.

[288] Ndumu, D.T. and Nwana, H.S., *Research and Development Challenges for Agent-Based Systems*, IEE Proceedings Software Engineering, Vol. 144, No. 1, pp. 2-10, February 1977.

[289] Neighbors, J.M., *The Draco Approach to Constructing Software from Reusuable Components*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 564-574, 1984.

[290] Neighbors, J.M., *The Evolution from Software Components to Domain Analysis*, International Journal of Software Engineering and Knowledge Engineering, Vol. 2., No.3, pp.325-354, May 1992.

[291] Nielsen, J., *Hypertext and Hypermedia*, Academic Press, 1990.

[292] Ning, J.Q., *A Component-Based Software Development Model*, In Proceedings Conference on Software Maintenance, pp. 389-394, 1996.

[293] Ning, J.Q., *Automated Support for Legacy Code Understanding*, Communications of the ACM, Vol. 37, No. 5, pp. 50-57, May, 1994.

[294] Norfolk, D., *The Book of Workgroup Computing*, Apricot, 1994.

[295] Notkin, D.S., *The Gandalf Project*, Journal of Systems and Software, Vol.5, No. 2, pp. 91-105, May 1985.

[296] Nwana, H.S., *Software Agents: an Overview*, The Knowledge Engineering Review, Vol. 11., No. 3., pp. 205-245, 1996.

[297] O'Donovan, B. and Grimson, J.B., *A Distributed Version Control System for Wide Area Networks*, Software Engineering Journal, September 1990.

[298] Oman, P.W. and Cook, C.R., *The Book Paradigm for Improved Maintenance*, IEEE Software, Vol. 7, No. 1, pp. 39-45, January 1990.

[299] Osbourne, W.M., *Building and Sustaining Software Maintainability*, In Proceedings of Conference on Software Maintenance, pp. 13-23, Austin Texas, September 1987.

[300] Osborne, W., *Software Maintenance and Computers*, pp. 2-14, IEEE Computer Society Press, Los Alamitos, 1990

[301] Osborne, W.M. and Chikofsky, E.J., *Fitting Pieces to the Maintenence Puzzle*, IEEE Software, Vol. 7, No. 1, pp.11-12, January 1990.

[302] Paakki, J., Salminen, A. and Koskinen, J., *Automated Hypertext Support for Software Maintenance*, The Computer Journal, Vol. 39, No. 7, pp577-597, 1996.

[303] Padula, A., *Use of a Program Understanding Taxonomy at Hewlett-Packard*, In Proceedings, 2nd Workshop on Program Comprehension, Los Alamitos, pp. 67-70, IEEE Computer Society, 1993.

[304] Palmer, J.D. and Fields, N.A., *Computer-Supported Cooperative Work*, Computer, pp.15-17, May 1994.

[305] Parnas, D.L., *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, Vol. SE-2, pp. 341-361, March 1976.

[306] Parnas, D.L., *Software Ageing*, In Proceedings 16th International Conference on Software Engineering, pp. 279-287, 1994.

[307] Parnas, D.L., *The Professional Responsibilities of Software Engineers*, In IFIP Proceedings, 1994.

[308] Paulk, M.C. and Konrad, M.D., *ISO Seeks to Harmonise Global Efforts in Software Process Management*, Computer, pp. 68-70, April 1994.

[309] Pennington, N., *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Cognitive Psychology, Vol. 19, pp. 295-341, 1987.

[310] Perry, D.E. *Software Interconnection Models*, in Proceedings 9th International Conference on Software Engineering, Monterey, California, pp. 61-69, March 1987.

[311] Perry, D.E., *Version Control in the Inscape Environment*, In Proceedings 9th International Conference on Software Engineering, Monterey, California, pp. 142-149, March 1987.

[312] Perry, D.E., *Inscape Environment*, In Proceedings 11th International Conference on Software Engineering, pp. 2-12, May 1989.

[313] Perry, D.E., *Legacy Architecture*, in Summary of the Dagstuhl Workshop on Software Architecture, Software Engineering Notes, Vol. 20, No. 3, pp. 63-83, July 1995.

[314] Pirie, I., *Benefits of Automating Configuration Management*, in Notes 1st Software Maintenance Workshop, Durham, England, 1987.

[315] Pressman, R.S., *Software Engineering - a Practitioners Approach*, 5th European Edition, McGraw Hill, 1997.

[316] Price Waterhouse, *Technology Forecast 1996*, version 6, Price Waterhouse, October 1995.

[317] Prieto-Diaz, R. and Neighbors, J., *Module Interconnection Languages*, Journal of Systems & Software, Vol. 6, No. 4, pp. 307-334, November 1986.

[318] Prieto-Diaz, R. and Freeman, P., *Classifying Software for Reusability*, IEEE Software, January 1987.

[319] Prywes, N., *Recovering Design and Specifications from Source Code*, IEEE Software, Vol. 13, No. 6, pp. 109-114, November 1996.

[320] Radice, R.A., Roth, N.K., O'Hara, A.C. and Ciarfella, W.A., *A Programming Process Architecture*, IBM Systems Journal, Vol. 24 (2), pp79-90, 1985.

[321] Ramadhan, H.A., *Improving the Engineering of Model Tracing Based Intelligent Program Diagnosis*, IEE Proceedings Software Engineering, Vol. 144, No.3, pp. 149-161, June 1997.

[322] Ramamoorthy, C.V. and Tsai, W-T., *Advances in Software Engineering*, Computer, Vol. 29, No. 10, pp. 47-58, 1996.

[323] Ray, R.J., *Experiences with a Script-Based Software Configuration Management System*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995

[324] Redmill, F., *Are your Systems Safety Critical?*, IEEE Review, pp. 93-96, May 1997.

[325] Redmill, F., *Technologies for Software Safety*, Computing and Control Engineering Journal, Vol. 9, No. 1, pp. 2-3, February 1998.

[326] Reddy, R., *The Challenge of Artificial Intelligence*, Computer, Vol. 29, No. 10, pp. 86-99, October 1996.

[327] Redmond-Pyle, D., *Graphical User Interface Design and Evaluation: a Practical Process*, Prentice Hall, 1995.

[328] Redmond-Pyle, D., *Software Development Methods and Tools: Some Trends and Issues*, Software Engineering Journal, pp. 99-103, March 1996.

[329] Reghabi, S. and Wright, D.G., *Software Engineering Release System (SERS)*, International Workshop on Software Version and Configuration Control, Grassau, pp. 244-263, January 1988.

[330] Reichenberger, C., *Delta Storage for Arbitrary Non-Text Files*, in Proceedings, 3rd International Workshop on Configuration Management, Trondheim, Norway, ACM Press, June 1991.

[331] Reichenberger, C., *Concepts and Techniques for Software Version Control*, Software – Concepts and Tools, Vol. 15, No. 3, pp. 97-104, 1994.

[332] Reichenberger, C., *VOODOO a Tool for Orthogonal Version Management*, ICSE SCM-4 and SCM-5 Selected Papers, Lecture Notes in Computer Science, ED. J. Estublier, Springer-Verlag, 1995.

[333] Render, H.S. and Campbell, R.H., *CLEMMA The Design of a Practical Configuration Librarian*, Proceedings Conference on Software Maintenance, Phoenix Arizona, pp. 222-228, October 1988.

[334] Roberge, J. and Smith, G., *Introduction to Programming in C++: a Laboratory Course*, Heath & Company, 1995.

[335] Rochkind, M.J., *The Source Code Control System*, IEEE Transactions on Software Engineering, Vol DE-1, No. 4, pp. 364-370, December 1975.

[336] Rohrbach, R. and Seiwald, C., *Galileo: A Software Maintenance Environment*, International Workshop on Software Version and Configuration Control, Grassau, pp. 444-456, January 1988.

[337] Royce, W.W., *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings of Wescon, August 1970.

[338] Ryan, N. and Smith, D., *Database Systems Engineering*, International Thomson Press, 1995.

[339] Saiedan, H. and Kuzara, R., *SEI Cabability Maturity Model's Impact on Contractors*, Computer, pp. 16-26, January 1995.

[340] Sakthivel, S., *A Decision Model to Choose between Software Maintenance and Software Redevelopment*, Journal of Software Maintenance: Research and Practice, Vol. 6, No. 3, pp. 121-143, 1994.

[341] Saywer, P. and Sommerville, I., *Direct Manipulation of an Object Store*, Software Engineering Journal, pp. 214-222, November 1988.

[342] Scacchi, W., *Managing Software Engineering Projects*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, pp, 49-58, January 1984.

[343] Schmerl, B.R. and Marlin, C.D., *Designing Configuration Management Facilities for Dynamically Bound Systems*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[344] Schmid, H.A., *Creating Applications from Components: A Manufacturing Framework Design*, IEEE Software, Vol. 13, No. 6, pp. 67-75, November 1996.

[345] Schneberger, S.L., *Software Maintenance in Distributed Computer Environments: System Complexity Versus Component Simplicity*, pp. 304-313, in Proceedings International Conference on Software Maintenance, Nice, France, 1995.

[346] Schneider, K. *Prototypes as Assets, not Toys – Why and How to Extract Knowledge from Prototypes*, In Proceedings Conference on Software Maintenance, pp. 522-531, 1996.

[347] Schreiber, G., Wielinga, B., de Hoog, R., Akkermans, H. and Van de Velde, W., *CommonKADS: A Comprehensive Methodology for KBS Development*, IEEE Expert, pp. 28-36, December 1994.

[348] Shapiro, S., *Splitting the Difference: the Historical Necessity of Synthesis in Software Engineering*, IEEE Annals of the History of Computing, Vol. 19, No. 1, pp. 20-54, 1997.

[349] Shephard, G., *CSCW - A Stage Beyond Groupware*, in: Computer Support for Co-operative Work, eds., Spurr, K., Layzell, P., Jennison, L. and Richards, N., pp. 195-216, Wiley, 1995.

[350] Sherer, S.A., *Cost Benefit Analysis and the Art of Software Maintenance*, In Proceedings Conference on Software Maintenance, pp. 70-77, Orlando, Florida, November, 1992.

[351] Siddiqi, J. and Shekalan, M.C., *Requirements Engineering: the Emerging Wisdom*, IEEE Software, March, 1996.

[352] Sloan, A., *Multimedia Communication*, McGraw Hill, 1996.

[353] Sloman, M., Kramer, J and Magee, J., *The Conic Toolkit for Building Distributed Systems*, 6th IFAC Distributed Computer Control Systems Workshop, Monterey, California, Pergamon Press, 1995

[354] Sneed, H.M., *Estimating the Costs of Software Maintenance Tasks*, In Proceedings Conference on Software Maintenance, pp. 168-181, 1995.

[355] Sneed, H.M., *Planning the Reengineering of Legacy Systems*, IEEE Software, pp. 24-34, January 1995.

[356] Softool Corporation, *CCC Change and Configuration Control Environment, Functional and Implementation Overview*, 1988.

[357] Softool Corporation, *Configuration Control CCC/DM Turnkey Dependency and Build Option, Users Manual*, 1989.

[358] Soloway, E., Adelson, B. and Ehrlich, K., *Knowledge and Processes in the Comprehension of Computer Programs*, in The Nature of Expertise, Chi., M., Glaser, R. and Farr, M., eds., Lawrence Erlbaury Associates, 1988.

[359] Sommerville, I., *Software Engineering*, 4th Edition, Addison-Wesley, 1992.

[360] Sommerville, I. and Dean, G., *PCL: a Language for Modelling Evolving System Architectures*, Software Engineering Journal, pp. 111-121, March 1996

[361] Sommerville, I., *Software Engineering*, 5th Edition, Addison-Wesley, 1996.

[362] Subramanian, S. *CRUISE: Using Interface Hierarchies to Support Software Evolution*, Proceedings Conference on Software Maintenance, Phoenix Arizona, pp. 132-142, October 1988.

[363] Sullivan, K.J., *Rapid Development of Simple, Custom Program Analysis Tools*, In Proceedings 4th Workshop on Program Comprehension, pp.40-44, Berlin, Germany, March 1996.

[364] Sun Microsystems, *Make User's Guide*, 1990.

[365] Swanson, E.B. and Beath, C.M., *Departmentalization in Software Development and Maintenance*, Communications of the ACM, Vol. 33, No. 6, pp. 658-667, June 1990.

[366] Swinehart, D.C., Zellweger, P.T., Hagmann, E.B., *The Structure of Cedar*, In Proceedings ACM SIGPLAN 85 Symposium on Language Issues in Program Environments, pp. 230-244, July 1985.

[367] Sycara, K, Pannu, A., Williamson, M. and Zeng, D., *Distributed Intelligent Agents*, IEEE Expert, pp. 36-45, December 1996.

[368] Takang, A.A. and Grubb, P.A., *Software Maintenance Concepts and Practices*, International Thomson Computer Press, 1996.

[369] Teitelman, W., *A Tour Through Cedar*, IEEE Software, Vol. 1, pp. 44-73, April 1984.

[370] Thomas, M., *The Impact of Computers on Society*, Seminar given at The University of Reading, December 1996.

[371] Tichy, W.F., *Software Development Control Based on Systems Structure Description*, Ph.D. Thesis, Carnegie-Mellon University, Computer Science Department, January 1980.

[372] Tichy, W.F., *Design, Implementation and Evaluation of a Revision Control System,* In Proceedings 6th International Conference on Software Engineering, Tokyo, Japan, pp. 58-67, September 1982.

[373] Tichy, W.F., *RCS - A System for Version Control*, Software Practice and Experience, Vol. 15, No. 7, pp 637-654, July 1985.

[374] Tichy, W.F., *Smart Recompilation*, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 3, pp. 273-291, July 1986

[375] Tichy, W.F., *Response to R.W. Schwanke and G.E. Kaisers Smarter Recompilation*, ACM Transactions on Programming Languages and Systems, Vol. 10, No. 4, pp. 633-634, October 1988.

[376] Tichy, W.F., *Tools for Software Configuration Management*, International Workshop on Software Version and Configuration Control, pp. 1-20, Winkler, J.F.H. ed., Grassau, Germany, January 1988.

[377] Tichy, W.F., *What is a Configuration?*, Summary of Plenary Discussion International Workshop on Software Version and Configuration Control, Grassau, pp. 169-171, January 1988.

[378] Tilbury, A.J.M., *Configuration Control of the Iterative Development Process.* In Proceedings Software Tools 89, London, pp. 103-113, June 1989.

[379] Tilley, S.R., *Domain-Retargetable Reverse Engineering III: Layered Modelling*, in Proceedings International Conference on Software Maintenance, pp. 52-61, Nice, France, October 1995.

[380] Tran, V., Liu, D-B. and Hummel, B., *Component-Based Systems Development: Challenges and Lessons Learned*, In Proceedings, Eighth International Workshop on Software Technology and Engineering Practice, pp. 452-462, London, July 1997.

[381] Tryggeseth, E., Gulla, B. and Conradi, R., *Modelling Systems with Variability using the PROTEUS Configuration Language*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[382] Umeda, Y. and Tomiyama, T., *Functional Reasoning in Design*, IEEE Expert, pp. 42-48, March-April 1997.

[383] van der Hoek, A., Heimbigner, D and Wolf, A.L., *A Generic Peer-to-Peer Repository for Distributed Configuration Management*, in Proceedings Conference on Software Maintenance, pp 308-317, Berlin, Germany, March 1996.

[384] Vatto, K., Yairi, R. and Palkama, M., *Integration of an ODBMS into Legacy Telecommunication Systems*, Object Expert, Vol. 1, No. 3, pp. 39-41, March-April, 1996.

[385] Venkatramani, K. and Clark, R., *Using Configured Directories to Solve Software Maintenance Problems*, Proceedings Conference on Software Maintenance, Phoenix Arizona, pp. 172-177, October 1988.

[386] Viskari, J., *A Rationale for Automated Configuration Status Accounting*, in Lecture Notes in Computer Science - Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Springer Verlag, 1995.

[387] von Mayrhauser, A. and Vans, A.M., *Industrial Experience with an Integrated Code Comprehension Model*, Software Engineering Journal, Vol. 10, No. 5, pp. 171-182, September 1995.

[388] von Mayrhauser, A. and Vans, A.M., *Program Comprehension During Software Maintenance and Evolution*, IEEE Computer, pp.44-55, August, 1995.

[389] von Mayrhauser, A. *Identification of Dynamic Comprehension Processes During Large Scale Maintenance*, IEEE Transactions on Software Engineering, Vol. 22, No. 6, pp. 424-437, June, 1996.

[390] Ward, M., *Transforming a Program into a Specification*, Centre for Software Maintenance Report 88/1, Unversity of Durham.

[391] Warwick, K., *Neural Networks: an Introduction*, Chapter 1 in: Neural Networks for Control and Systems, pp. 1-11, Eds. Warwick, K, Irwin, G.W. and Hunt, K.J., IEE Control Engineering Series, Peter Peregrinus, 1992.

[392] Warwick, K., *An Introduction to Control Systems*, 2nd Edition, Advanced Series in Electrical and Computer Engineering, Vol. 8, World Scientific, 1996.

[393] Wasserman, A.I., *Toward a Discipline of Software Engineering*, IEEE Software, Vol. 13, No. 6, pp. 23-31, November 1996.

[394] Welsh, T. *Vax DEC/CMS* A Practical Configuration Management Tool*, In: Proceedings Software Tools 89, London, England, June 1989.

[395] Whitley, D.J., *The Benefits of Automated Configuration Management*, Scicon Limited.

[396] Whittle, B., *Models and Languages for Component Description and Reuse*, ACM SIGSOFT, Software Engineering Notes, Vol. 20. No. 2, pp. 76-88, April 1995.

[397] Wilkinson, K., Lyngbaek, P. and Hasan, W., *The Iris Architecture and Implementation*, IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.

[398] Winkler, J.F.H., *The Integration of Version Control into Programming Languages*, Lecture Notes in Computer Science 224, Proceedings International Workshop Configuration Management, pp. 230-250, Trondheim, Norway, June 1986.

[399] Winkler, J. F.H., *Version Control in Families of Large Programs*, Proceedings 9th International Conference on Software Engineering, Monterey, California, March 1987.

[400] Winkler, F.H. and Stoffel, C., *Program-Variations-in-the-Small*, International Workshop on Software Version and Configuration Control, Grassau, pp. 175-196, January 1988.

[401] Wong, K., Tilley, S.R., Muller, H. and Storey, M-A.D., *Structural Redocumentation: A Case Study*, IEEE Software, pp. 46-54, January 1995.

[402] Wright, D.G., *Configuration Management in a Heterogeneous Environment*, In Proceedings Conference on Software Maintenance, Phoenix, Arizona, pp. 218-221, October 1988.

[403] Yau, S.S. and Tsai, J.J., *Knowledge Representation of Software Component Interconnection Information for Large-Scale Software Modifications*, IEEE Transactions on Software Engineering, Vol. SE-13, No.3, pp. 355-361, March 1987.

[404] Yeo, B-L. and Yeung, M.M., *Retrieving and Visualising Video*, Communications of the ACM, Vol. 40, No. 12, pp. 43-52, December 1997.

[405] Yip, S.W.L., *Software Maintenance in Hong Kong*, in Proceedings International Conference on Software Maintenance, pp. 88-95, October 1995, Nice, France, IEEE Computer Society Press, 1995.

[406] Yourdon, E., *A Tale of Two Futures*, IEEE Software, pp. 23-29, January-February, 1998.

Glossary

Abstraction:	A technique for handling complexity by masking irrelevant details for defined situations.
Adaptive maintenance:	Changes made to both the design and code in order to enable the software product to accommodate changes in its hardware and software operating environments.
Alien components:	Components that are never referenced by the application system and as such play no part in the system configuration at any level of abstraction.
Allied components:	The general system libraries and other operating system components that are required by an application, but which are not actually maintained as part of it.
Anomalous configuration abstraction:	Modelling of a system configuration which has become corrupted or masked through the presence of alien components, or incomplete through the loss or misplacement of essential components.
Application components:	The core components of the baseline of an application.
Associated components:	The components of a system such as the application language libraries and reusable components that are associated with the application under development or maintenance.
Attribute:	A property of a particular software configuration component.
Baseline configuration abstraction:	Models the traditional view of a software system configuration, that is, the core component parts of a software application.

Baseline:	A stable, unchanging or 'frozen' version of a configuration. It provides the inventory of the state of a product at a specific point in time and serves as a basis for subsequent development, control and maintenance of a system
Change control:	The process that ensures that changes to individual software components are made and incorporated in the correct fashion. It controls what changes are made and when, where and by whom the change are made.
Change request:	A document or record that proposes a change. If arising from an error in the software it may also be referred to as a problem report.
Change-log:	Documents the history of a component (see Derivation).
Cognitive components:	The components of a system that record the incrementally gathered information about the functionality, structure and dependencies about the system.
Coincidental utilities:	Tools that are intrinsically available on the host system environment, which although not specifically provided for SCM purposes can be used to provide information regarding software system configurations.
Component attribute set:	The full complement of information about a particular configuration component.
Computer aided software engineering (CASE)	Any computer software or system which aids a software engineer in the specification, design, development, testing or maintenance of other computer software or any aspect of the management of this process.
Configuration abstraction:	A subset of component types, modelled according to a defined pattern, in order to identify all or part of a software system configuration.
Configuration control:	The management of complete versions of systems and the interrelationships existing among the components of the system.

Configuration history	The time-related series of genetic fingerprints which progressively record or reclaim the evolutionary characteristics of a particular software system.
Configuration validation:	Process of ensuring that the SCIs serve their correct purpose, i.e. that they meet customer requirements.
Configuration verification:	Process of ensuring that the specification for each SCI in one baseline or update is achieved in the subsequent baseline or update.
Configuration:	A set of interrelated software objects from which the system is composed.
Corrective maintenance:	Changes made to code resulting from inconsistencies between the specification of the product and the product itself.
Data components:	The data file components of a system, such as the database or spreadsheet data that are consumed and produced as a result of running the application components.
Delta	Space efficient storage mechanism for versions of components.
Dependency analysis:	Scrutiny of the program code in order to ascertain the relationships existing between the particular module to be changed and the other parts of the system (see Impact Analysis).
Derivation:	Records precisely and accurately the changes made to a software object, including when, where and by whom the change was made.
Derived object:	A software object that is created automatically by program or tools called derivers.
Documentary components:	The formally produced requirements, design etc. documentation that supports the development of a system.
Documented configuration abstraction:	Model of the relationships between the baseline application and any documentation developed during the development or maintenance processes.

Domain analysis:	The process of understanding the application context and implementation of working practices within the product.
Domain components:	The components of a system that contain documented or cognitive knowledge regarding the domain environment or application area in which the system is designed to operate.
Enabling components:	Components of a system that do not form part of either the core or secondary software system but which are tools to help with the control or program comprehension of the software application.
Environmental components:	Components such as compilers, linkers, editors, tools and hardware that make up the operational environment of the system.
Environmental configuration abstraction:	Models the interactions occurring between the application components and those of the surrounding primary or secondary environment.
Existing system:	Software system that has entered the maintenance phase. Also synonymous with operational and legacy systems.
Extensible system information base (ESIB):	Information base containing data regarding key features of operational environments such as operating systems, systems architectures, application types, tools and programming as well as the reclaimed information about software system configurations.
Firmware:	A hardware device and the software that resides on that device, where the software cannot be readily modified under program control.
Forward engineering:	The traditional approach to software development, moving from analysis through design to implementation and testing.
Function tools:	Tools that support one aspect of the software configuration management process, for example version control or system synthesis but generally not both without forced integration.

Generic configuration:	A system model, which makes possible the compact representation of a large number of baseline configurations.
Genetic fingerprint	The unique description of a software system configuration at a particular defined point in time.
Geriatric systems:	Legacy systems developed longer than 10 years ago.
Green-field softwarê systems:	Systems developed from new, thereby enabling software configuration management principles and techniques, together with other modern programming, design and management methods and practices to be employed on the system.
Heterogeneous system	Software system composed of a wide variety of different component types.
Homogeneous system	Software system composed of a single (or very few) component type(s).
Hypermedia:	Non-linear linkages between and within files of different media types, often of a non-linear nature.
Hypertext:	Non-linear linkages between and within text or graphic files, often of a non-linear nature.
Impact analysis:	Scrutiny of the program code in order to ascertain the 'what if' effects of changing a particular module on the other parts of the system (see Dependency Analysis).
Incremental documentation:	Retrospective detailed documentation conducted during maintenance, of the parts of the system requiring maintenance.
Integrated project support environments (ipse):	Environments that manage the complete project lifecycle and offer project management support facilities that go beyond those of configuration management.

Inter-application modelling:	Modelling of a system in connection with the identification of (inter-specific) relationships existing between components of the core application configuration and components other types.
Interconnection model:	A model proposed to support the management of system evolution.
Interface components:	The interface components of a system, either developed in-house or reused from a library of interface components and widgets.
Inter-specific relationships:	Relationships existing between the components of an application and the other identified component types.
Inter-version-group control:	Management of changes to individual components such that ther version group is extended.
Intra-application modelling:	Modelling of a system in connection with the identification of (intra-specific) relationships existing between application components of the core configuration.
Intra-specific relationships:	Relationships existing between the components within an application at a source code level.
Intra-version-group control:	Management of changes to configurations such that the program family group is extended.
Inverse Configuration Description Language:	Language developed to model system architectures and configurations at a programming-in-the-environment (PITE) level.
Inverse engineering:	The process concerned with the complete reverse engineering of a system from code to specification through the use of formal transformations.
Inverse software configuration identification (ISCI):	The process of re-identifying and documenting the existent but often corrupted configuration of an existing software system with a view to bringing it back under configuration control.
Inverse software configuration management (ISCM):	The process of bringing an existing software system back under configuration control.

ISCM maintenance process model:	The conceptual stages/activities necessary to effect the ISCM process, and its relationship to the global software lifecycle.
ISCM process:	The overall process of Inverse Software Configuration Management and its definition.
Legacy system:	Application software created in a pervious traversal of a software lifecycle.
Library:	A common and controllable store for the elements of a software system that provides a basis for sharing and control of software objects.
Location:	The physical storage position of a particular software configuration component.
Maintenance process model:	A coherent and defined set of conceptual activities carried out during maintenance.
Master configuration index:	A list that uniquely identifies all configuration items of a system.
Method:	The set of defined activities required to physically realise the abstract model of a process.
Missing components:	These are components that are referenced by the system but which have been lost, destroyed or misplaced.
Model:	An abstract representation of a process.
Module interconnection language (MIL):	A language that describes the structure and evolution of software systems.
Multimedia:	A combination of more than one media format such as text, graphics, audio, video, animation.
Object base:	The underlying file or database structure within which to store the various software products and configurations.

Operational system:	Software system that has entered the maintenance phase. Synonymous with existing and legacy systems.
Out of control system:	Legacy software system developed without the aid of software configuration management principles and techniques such that it is characterised by high program comprehension overheads and error prone maintenance.
Perfective maintenance:	Changes made to the specification, design and code in connection with improving the function of the software in response to user requests for improvement, enhancements to functionality, or customisation to new working practices.
PISCES M^4 system:	The meta-CASE environment developed to semi-automate the initial stages of the ISCM model and PISCES method.
Pre-emptive maintenance:	Activities carried out during development whose primary aim is to reduce future maintenance costs.
Preventive maintenance:	Changes made to the software in order to preclude future problems and facilitate future maintenance work.
Primary environment:	The components with which the application components have a direct relationship in terms of their required integration or interaction to enable execution of the software.
Problem report	A document that describes a fault within the system (see Change Request).
Process architecture:	The modelling tool used to represent the process model at varying levels of abstraction.
Proforma identification complexity series (PICS):	Series of proformas or templates developed to guide the information collection and collation process during the reclamation of software system configurations.

Proforma identification scheme for configurations of existing systems (PISCES) method:	The physical realisation of each of the above stages / activities through a defined sequence of steps and templates.
Program comprehension:	The process of understanding the application system prior to making a modification. Although often used synonymously with the term system comprehension, it is used in the context of this thesis to mean understanding at the source code level.
Program family configuration abstraction:	Models versioning of a system at application baseline, release, primary or secondary environmental levels of detail.
Program family:	A set of closely related but distinct software systems, each member catering for slightly different requirements.
Programming-in-the-environment (PITE):	The process concerned with the interactions between modules (components) of a system and its environment. Developed as an extension to the concepts of PITS and PITL.
Programming-in-the-large (PITL):	The process concerned with the interactions between modules (components) of a system.
Programming-in-the-small (PITS):	The process concerned with the development of an individual module (component) of a software system.
Redocumentation:	The process of studying the code in order to create an alternative, easy to visualise, representation of the program structure.
Redundant components:	Components that are never referenced by the application system (see alien components).
Re-engineering:	The process of reverse engineering followed by a period of forward engineering during which additional functionality may be added to the original system.

Relationship:	The nature of the bond existing between two components within a group (intra-application relationships) or between components in different groups (inter-application relationships)
Restructuring:	The process of transforming the representation of the system from one form to another at the same relative level of abstraction.
Reverse engineering:	The process that combines an element of redocumentation with that of design recovery such that it enables a system to be represented at a higher level of abstraction.
Revision:	A source object produced by changing another source object. Revisions are sequential and cumulatively record development history such that later versions supersede earlier versions.
Ripple effect:	Propagation of errors through a software system as a result of making a change to that software system.
Secondary environment:	Those components which enable construction or change of the baseline application but which are not actually maintained or incorporated as part of it.
Software architecture:	This is a more refined representation of a software system than a model. It is still conceptual and logical in nature but deals with a lower level of granularity in terms of describing particular abstract representations of a system.
Software configuration abstraction	A model of the software configuration from a particular viewpoint or defined level of granularity.
Software configuration audit:	The process of determining whether or not baselines meet their requirements and that correct procedures are being adhered to.
Software configuration component:	This is the fundamental building block of an application system. It is at the level of an individual physical item within a system, i.e. not a procedure or function within a program or module (see also software configuration item).

Software configuration control:	The set of controlled procedures for making changes to components and baselines.
Software configuration identification:	The unique identification and definition of the different components and associated baselines of a system, and any changes made to these components and baselines.
Software configuration item (SCI):	The basic unit of information within a configuration, examples include source code files, object code files, command files, database files, documentation files, test procedures.
Software configuration management (SCM):	The process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system lifecycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items.
Software configuration management systems:	Offer in a single product all aspects of SCM, i.e. seamless integration between identification, control, status accounting and audit.
Software configuration status accounting:	Provision of an administrative history of the evolution of a software system.
Software configuration:	This is the physical combination of a set of interrelated components (SCIs) that together form a viable system. In essence it is the physical realisation of a particular software architecture. The combination may be at varying levels of abstraction and in particular at baseline, family or environmental.
Software engineering lifecycle:	A software lifecycle that encompasses representation of both the technical activities and management aspects associated with software development and maintenance.
Software lifecycle:	Cyclic ordering of the activities defined within a software process model.

Software maintenance:	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 model:	The overall abstract representation of a software system. It is essentially a conceptual model defining the entire set of components from which a software system may be composed and the possible relationships existing between the component types (often used synonymously with system model).
Software object:	Any identifiable, machine-readable document, the basic unit of information within a configuration (see Software Configuration Item).
Software process model:	A series of activities associated with the development or maintenance of a software system.
System comprehension:	The process of understanding the application system prior to making a modification. Although often used synonymously with the term program comprehension, it is used in the context of this thesis to mean understanding the interactions of the source code with the rest of its system environment.
System evolution:	The process by which a program family evolves.
System model:	This is the overall abstract representation of a software system. It is essentially a conceptual model defining the entire set of components from which a software system may be composed and the possible relationships existing between the component types (often used synonymously with software model).
Third party components:	Components such as database software and other proprietary software that belongs outside the application but with which it interacts.
Time-space divide:	The temporal and/or geographic separation of development and maintenance activities and/or personnel associated with the hand-over of a system from developer to client.

Total configuration abstraction:	Models an entire application and thus contains components in any or all of the identified components groups.
User components:	The user of a system.
Variant:	A source object created by changing another source object. Variants are indistinguishable under a given abstraction and as such are intended to be alternative, interchangeable parts.
Version	A new software configuration item (component) arising from changes made to existing configuration items as a result of the need to correct, adapt or enhance the software. Versions may be revisions or variants.
Version control:	Management of version groups arising from changes to the individual components of a system.
Version group	The collective set of versions arising from changes to a component.
Versioned components:	The variants and revisions of a system that result from changing components in a system.