The Use of Non-Formal Information in Reverse Engineering and Software Reuse

A thesis submitted for the degree of

Doctor of Philosophy

by

Alan J. Brown

Department of Computer Science, Brunel University

1992

Abstract

Within the field of software maintenance, both reverse engineering and software reuse have been suggested as ways of salvaging some of the investment made in software that is now out of date. One goal that is shared by both reverse engineering and reuse is a desire to be able to redescribe source code, that is to produce higher level descriptions of existing code.

The fundamental theme of this thesis is that from a maintenance perspective, source code should be considered primarily as a text. This emphasizes its role as a medium for communication between humans rather than as a medium for human-computer communication. Characteristic of this view is the need to incorporate the analysis of non-formal information, such as comments and identifier names, when developing tools to redescribe code. Many existing tools fail to do this.

To justify this text-based view of source code, an investigation into the possible use of non-formal information to index pieces of source code was undertaken. This involved attempting to assign descriptors that represent the code's function to pieces of source code from IBM's CICS project.

The results of this investigation support the view that the use of non-formal information can be of practical value in redescribing source code. However, the results fail to suggest that using non-formal information will overcome any of the major difficulties associated with developing tools to redescribe code. This is used to suggest future directions for research.

Table of Contents

Acknowledgements

I would like to thank my two supervisors during this project; Professor P.A.V. Hall now of The Open University, and Professor L. Johnson of Brunel University for their guidance and insight. I would also like to thank the members of IBM UK (Hursley) who have been involved with this project, in particular Mark Phillips and Pete Collins.

I am also indebted to many of the members of staff in the Department of Computer Science at Brunel, and also to the research students and support staff who have helped in the course of this project. A special thanks must be given to the members of staff of CRICT (Centre for Research in Information Culture and Technology), and in particular Dr Janet Low, who helped provide me with much needed direction.

Finally, I would like to thank my parents and my sister for their enduring support not only during this project but throughout the whole of my life.

The work reported in this thesis was jointly funded by SERC and IBM UK Labs (Hursley) as CASE award, Ref No. 88503626.

Chapter 1
Introduction

1.1 Background

The original brief of this project was to develop tools and techniques to abstract Z specifications from existing code. The motivation behind such a reverse engineering project was clear. IBM UK (Hursley) who helped fund this work had an ongoing project to respecify much of the code which makes up their CICS product in the formal specification language Z.

CICS is a large (about 750,000 lines) product which has been in development for over twenty years. It provides an applications programmer with a suite of programs with which to build transaction processing systems.

This work to respecify parts of CICS was (and still is) being carried out to produce up to date documentation for existing code that may have been written many years earlier. This new documentation increases the maintainability of the code for which it has been developed thus making it easier to modify, and also brings the documentation of old code into line with the documentation standards of newly produced code.

It soon became apparent that there was to be no magical philosophers stone for transforming dowdy, base source code into shining new (and much more valuable) formal specifications. However, there did seem to be considerable room in which to develop tools for analysing source code and extracting predicates and invariants which may eventually form part of a formal specification, or similarly tools for replacing sections of source code (that were congruent with certain predefined forms) with higher level descriptions of that code.

The difficulty of the task of producing Z specifications from source code seemed not to be directly related to the problems of analysing source, but seemed more to do with the huge gulf in what is often termed "the level of abstraction" between the source code and the specification that was to be derived from it. This gap between code and specification was the one that needed bridging if reverse engineering was to be successful, and yet, very little of the current work on reverse engineering tools seemed to accept the enormous difference between these two levels.

In general most attempts at developing reverse engineering tools seemed to take the view that any new information or expression that we can derive from a piece of code will surely make the reverse engineering task easier, rather than considering what constituted the difference between these two levels of abstraction. This "more is better" approach was to be done without considering what role such information might play in the final specification. Predicates representing the action of code can be automatically generated from code, but this predicate in itself is useless unless it can be integrated into the framework of a formal specification which *as a whole* acts as a description of the original source code.

This frequently seemed to be the case in many approaches to what has been termed in this thesis *redescribing source code.* Redescribing source code aims semi-automatically to derive higher level descriptions of existing code, for example a tool which semi-automatically derives Z specifications from CICS source code.

Although the importance of comments and documentation in understanding source code is well documented, nearly all the approaches to redescribing source code seemed to pay very little attention to using this *non-formal* information in their analysis of source code. Instead tools for redescribing source code seemed to concentrate exclusively on analysing the formal structure of the code.

This omission seemed at odds with the aims of redescribing source code. Frequently it is the non-formal information that is closely related to the high level operation of a piece of code, that is to the sort of operations that a formal specification might describe, rather than lower level features of the code's structure. This observation seemed to suggest that using non-formal information as part of a reverse engineering tool may well help to bridge the gap between code and specification.

The drawbacks of using non-formal information in code analysis though were quickly pointed out by other researchers, namely the inconsistency and unreliability traditionally associated with comments in code. To convince people of the value of such information it seemed necessary to achieve two goals. Firstly to theoretically argue not just for the potential utility of non-formal information, but for the *necessity* of using this information in redescribing source code, and secondly to demonstrate the practicality of using non-formal information in tools for redescribing source code. It is these two goals that this thesis seeks to accomplish.

1.2 Overview of Thesis

This thesis is about highlighting a view of source code as being primarily a *text*. In particular it aims to highlight the potential role of non-formal information, characteristic of this textual viewpoint, in providing information about the nature of source code for use in tools to facilitate reverse engineering and software reuse.

Frequently within software engineering, talk about "the software crisis" or "the maintenance crisis" is heard. These terms refer to the increasing proportion of costs associated with software when developing computer systems, and to the increasing costs incurred in the maintenance of existing software. Perhaps one of the most alarmist expressions consistent with the idea of a maintenance crisis is the view that eventually a point will be reached where organisations will no longer be able to develop software because all the resources of the organisation are tied up in maintenance activities pressman

Among solutions to the growing problems associated with software are reverse engineering and software reuse. Reverse engineering aims to improve the maintainability of existing systems by redocumenting them, often in a style that was not used during the development of the original code (ie formal specifications or object-oriented languages). Software reuse aims to reduce development costs by reusing code and other artifacts of software development such as designs and specifications in the development of new systems.

Both reverse engineering and software reuse have a need to be able to develop high level descriptions of existing source code (ie specifications and designs). Whilst software reuse has mainly restricted itself to finding a suitable representation with which to describe source code components, research on reverse engineering has produced a number of approaches which attempt to automatically or semi-automatically produce high level descriptions from existing source code. These approaches attempt to redescribe source code in a new form. Ultimately, the aim of this work is to facilitate the production of designs and specifications of existing code where these documents are either missing or inaccurate.

The main argument presented in this thesis is that existing approaches to redescribing source code are biased far to heavily towards considering only the formal information contained in source code. That is the kind of information that relates directly to the compilation and eventual running of the code as opposed to considering the importance of non-formal information. This is the component of source code that is normally associated with improving the comprehensibility of code such as comments and the use of meaningful identifier names.

This argument is based around viewing existing approaches as having a formalist view of source code. They consider source code as defining a formal object. This formal object is identified with the 'meaning' of source code and serves as a basis for drawing inferences about the functionality of the source code.

Whilst such a view works well for the needs of code optimisation and for program validation, the aims of producing a higher level description of the original code render this formalist viewpoint as too restrictive since it is unable to account for factors which affect the comprehensibility of source code.

The view expressed in this thesis is that source code should be seen primarily as a *text* rather than as a formal object. This enables non-formal features of source code to be considered as an integral part of the text rather than as annotations of the formal object defined by source code. Considering source code as a text is used to suggest some potentially useful methods of analysing source code, consistent with the needs of reverse engineering and software reuse. One particular approach, based upon research done in information retrieval is investigated in an attempt to empirically demonstrate the potential usefulness of non-formal information in redescribing source code.

This experimental work aims to use the occurrence of terms within code as the basis for automatically indexing pieces of source code according to their function. A number of different indexing functions for the automatic indexing of source code were developed and evaluated using code from IBM's CICS product. Thus this experimental work was performed on commercially developed code. The results of this investigation demonstrate the potential usefulness of non-formal information in providing information about the high level goals of source code, and so adds support to the claim that approaches to redescribing source code should make more use of non-formal information.

The results also indicate that whilst utilising non-formal information may be of benefit, the use of non-formal information fails to overcome any of the major problems associated with redescribing source code. It is suggested this failure is due to the assumptions on which attempts to redescribe code are based because these fail to note the importance of the environment in defining the role of information systems.

1.3 Chapter Summaries

Chapter 2 introduces the field of software maintenance, and the view that software maintenance should not be seen as a process of 'correction' but as a process of program evolution. This program evolution is both necessary and unavoidable, however this leads to a general degradation of software quality. Reverse engineering is a means of slowing this degradation and increasing the lifetime of software by redocumenting existing software. Tools for reverse engineering are categorised and described. The field of software reuse is introduced as a means of improving the efficiency of software development and of salvaging parts of existing systems that have become obsolete. Finally the relationship between software reuse and reverse engineering is discussed.

Chapter 3 provides a more detailed discussion of the idea of redescribing source code as an approach to reverse engineering, describing the goals and characterising the approaches taken. This is then followed by describing existing approaches to automatically redescribing source code in more detail.

Chapter 4 aims to provide reasons for the necessity of using non-formal information in redescribing source code. It begins with a description of some of the problems faced by a formalist view of meaning. This is followed by an alternative view of meaning and representation based upon the interpretation of texts in general. Source code and other artifacts of software development are then viewed as texts and this leads to a critique of current approaches to redescribing source code. Ways are suggested in which non-formal information could be extracted from source code and used to supplement the analysis of code given by existing approaches.

Chapter 5 provides the background theory to an investigation intended to demonstrate the feasibility of using non-formal information in the analysis of source code. This investigates the use of natural language terms in source code as a basis for automatically indexing pieces of code according to their function. This chapter introduces the background theory from information retrieval and decision theory necessary for the experimental work.

Chapter 6 provides more specific details of the investigation. It describes how the investigation was performed and introduces the different indexing functions that were evaluated.

Chapter 7 presents the results of the applying the indexing functions developed in chapter 6 to previously unseen pieces of code. These results are then followed by a more detailed analysis of the general approach used and the way such results could be of practical use. This is then followed by describing possible extensions and some limitations of the approach.

Chapter 8 attempts to explain why a variety of approaches to redescribing source code, including the method investigated in this thesis, have failed to overcome the fundamental difficulties affecting this process. This leads to suggestions for the direction that future research should take.

Chapter 2
Software Maintenance, Reverse Engineering and Reuse

2.1 Software Maintenance

It is currently estimated that between 40% and 70% of all expenditure on software by organisations is concerned with activities usually covered by the term *software maintenance* (Foster, Jolly and Norris 1989). Given this large amount of expenditure, it is clearly necessary to ensure that maintenance activities are properly understood and managed effectively.

Software maintenance is something of a blanket term which tends to cover almost all alterations to software which occur after the software has been delivered, hence terms such as "enhancement", "adaptation", "support" and "further development" are used to describe maintenance activities. Usually, maintenance activities are classified into four categories (Pressman 1987):

1. Corrective Maintenance - diagnosis and correction of errors in the software.
2. Adaptive Maintenance - modification of software to cope with new environments.
3. Perfective Maintenance - modifications and enhancements of the software, usually in response to user requests.
4. Preventative Maintenance - improvements to the future maintainability of the system.

This characterisation of software maintenance suggests that maintenance is not simply an activity that occurs to correct errors introduced by the original development process, and hence could be eliminated by a sufficiently rigorous development process, but rather suggests that the need for maintenance is an intrinsic property of *all* software.

This view of maintenance and of software lead Lehman and Belady to use the term *program evolution* (Lehman and Belady 1985) to describe this ever changing nature of software. They argue that all software will need to undergo significant changes whilst in use, regardless of the original intentions of the developers. They encapsulate this view of software in their law of continuing change:

> "A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost-effective to replace the system with a recreated version."

As a consequence of this law, they also formulate the law of increasing complexity:

> "As an evolving program is continually changed its complexity, reflecting deteriorating structure, increases unless work is done to maintain of reduce it."

These two laws express both the inevitable need for software maintenance, and the inevitable degradation of software quality over time (from Lehman and Belady 1985, p412).

For many organisations, the software that they use constitutes a major investment and can be seen as a significant asset of the organisation. The law of increasing complexity suggests that over time, the value of this asset will depreciate.

Eventually, the software will reach a point where it has become unmaintainable. At this point the software has become either too difficult or too expensive to maintain. Corbi (Corbi 1989, p40) calls this phase of maintenance the *phase of obsolescence* and notes that the beginning of this phase may be caused by:

- The loss of the only person who understood the workings of an undocumented program.
- Inability to maintain the support software or hardware.
- The loss of the source code and other documentation through fire, flood, poor configuration management etc.

Although the software may still be used during this phase, the lack of maintenance ensures that eventually the software will become obsolete. At this point, the investment that the organisation has made in this software has become moribund. The software has ceased to be a company asset. Clearly, this loss of an asset, especially one that has had money invested in it over a long period of time, is undesirable.

Not only does the obsolete software represent an investment in a financial sense, but it can also be viewed as an investment in an intellectual sense. The software can be viewed as representing knowledge about the operation of the organisation that has been accumulated through its development. This is particularly true in the case of many applications systems where the system can clearly be seen as encapsulating knowledge about many of the processes that constitute the organisation itself.

It is therefore very important for organisations to prevent this stage of obsolescence being reached by software. If such a stage is reached for unavoidable reasons, it would be of great value to an organisation to be able to salvage something from the now defunct software.

Reverse engineering represents one way of delaying the onset of obsolescence and of salvaging already obsolete software, whilst one motivation for software reuse is to provide a mechanism for this salvaging since it allows some of the original investment made in software to be recovered in a realisable form. Thus both reverse engineering and software reuse aim to provide a partial solution to the problems of software maintenance.

2.2 Reverse Engineering

The degradation in the quality of software over time implied by the law of increasing complexity leads to the source code of a particular system becoming very difficult to maintain. As more alterations are made to code, the structure of the code tends to deteriorate making the code harder to understand. At the same time, documentation is frequently not kept up to date when these alterations are made and this further increases the difficulties of maintaining such code. Source code that has deteriorated in this way is often referred to as "old" even though it is not necessarily old in a chronological sense. Corbi (Corbi 1989, p298) lists six attributes which characterise old code:

1. Design was done with methods and techniques that do not clearly communicate the program structure, data abstractions and function abstractions.

2. Code was written with a programming language and techniques that do not quickly and clearly communicate the program structure, the program interfaces, data structures and types, and functions of the system.

3. Documentation is non-existent, incomplete or not current.

4. Design and code are not organised in such a way as to be insulated from changing external hardware or software.

5. Design was targeted to system constraints that no longer exist.
6. Code contains parts where nonstandard or unorthodox coding techniques were used.

The presence of any one of these attributes can make maintenance difficult. The presence of most or all of these can make maintenance extremely expensive and time consuming if not impossible. Any process that can either remove these features from software or ease the maintenance problems caused by them would significantly aid software maintenance.

Any activity which involves modifying the operation of existing source code can be divided into two phases. Firstly, the engineer performing the modification must come to an understanding of the operation of the software, then secondly, the alteration must be implemented. Although in practice these two phases are interlinked, with the nature of the alteration influencing the understanding that is arrived at, and the process of making the alteration leading to a still greater understanding, this distinction often proves useful when discussing software maintenance.

If we consider the attributes of old code listed above, (1), (2), (3), (6), and to a lesser extent (5) can all be seen as applying to the ease with which code can be understood, whilst only (4) and (6) can be seen as relating to the ease with which the modification itself can be implemented.

Further, a study of the process of modifying existing software has found that over half the time spent in this activity was related to program understanding (Fjeldstad and Hamlen 1983) rather than to actually implementing the modification. It seems reasonable to suggest that program understanding is a major component of maintenance activities.

Reverse engineering is directly aimed at providing a way of overcoming some of the problems associated with old code by providing up to date documentation for that code and so facilitating the task of program understanding. Chikofsky and Cross (Chikofsky and Cross 1990, p15) describe reverse engineering as:

the process of analysing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction

The most important characteristic of reverse engineering is that it is not to involve altering the source code *in any way*. Reverse engineering does not alter the operation of the system but is a process of examination and description.

This makes reverse engineering distinct from what is often termed *re-engineering* which does involve altering the structure of the source code. Often re-engineering involves an element of reverse engineering before the code is altered and is often related to adaptive maintenance, eg the porting of an existing system to a new environment. This thesis is not concerned with the process of re-engineering but concentrates primarily on the aims of reverse engineering. It is accepted that occasionally reverse engineering results in the discovery of errors in the original program which may then need to be fixed, however this is considered as a significant example of re-engineering.

It is important to make clear the goal of reverse engineering. The aim of reverse engineering is *to improve the future maintainability of a system*. The process of reverse engineering does not produce any immediate gain. The success (or otherwise) of reverse engineering a system is solely to be judged on the *future* ability to maintain a system. As a secondary goal, by providing documentation for existing code reverse engineering may also be used to identify and document code for software reuse. In this case the success of reverse engineering again depends upon the ability of the produced documentation to describe the code and allow it to be 'maintained' in a new environment.

The future maintainability of a system is clearly very hard to quantify, however it is important to keep this goal in mind when discussing reverse engineering. This goal does not necessarily entail that reverse engineering is somehow the "inverse" of forward development, neither does it mean that reverse engineering can be seen as an attempt to somehow uncover the "original" design of a system. It is often the case that one of the reasons for reverse engineering is to produce up-to-date documentation of a system in a style, such as an object oriented description, that was not in existence when the software was developed. Reverse engineering is a process of redocumentation.

This goal of improved maintainability places great emphasis on the *quality* of documentation that is produced in reverse engineering. It is clearly insufficient to automatically (or otherwise) produce a large volume of new documentation if that documentation is too unwieldy or too inaccurate to be used. In sum, this means that the decision to perform reverse engineering needs to be seen as a long term strategic decision, and to be managed as such. It is likely to be counter-productive to see reverse engineering as yet another "quick fix" that will hold a creaking system together for a short while longer.

2.3 Reverse Engineering Tools

Recently, there have been many tools that have been developed primarily with the aim of facilitating reverse engineering. Chikofsky and Cross (Chikofsky and Cross 1990) note two sub-areas of tools developed for reverse engineering that they term *redocumentation* and *design recovery.* Redocumentation is intended to refer to those tools that automatically generate new documentation from existing source code, such as pretty printers or cross-referencers. Design recovery aims to incorporate domain models and other external sources of information to reproduce all the information that is necessary for a person to understand the workings of a piece of code at the design level.

The characteristics of these distinctions are hard to discern, and as Chikofsky and Cross are aware, there is much research into reverse engineering tools that does not fit into these categories. Since many tools use a number of techniques, for example design recovery (in the above sense) is quite likely to involve the use of redocumentation tools, it seems more natural to categorise the main functional components that one finds in reverse engineering tools as opposed to attempting to categorise tools and systems *per se*.

There are four main functional components which are to be found in reverse engineering tools; program analysis tools, database repositories, user interface, and tools for redescribing source code. The last of these is to form the main focus of this thesis. I will now give a brief description of each of these components and provide some examples of systems which have been developed that concentrate on each of these areas.

2.3.1 Program analysis tools

These tools implement algorithms that allow information about the nature of source code to be extracted. Most commonly, they perform static analysis on the source code to allow control flow, data flow and cross referencing information to be obtained. More sophisticated approaches may implement symbolic execution algorithms or dynamic analysis on the source code. This information may then be used as a basis for modularising the code according to fixed criteria.

Many of these techniques derive from techniques originally used in the design of compilers, although some (modularisation) are more specifically concerned with the demands of reverse engineering. The algorithms they employ are characteristically "non-heuristic" and their goals well defined. As such, the algorithms themselves are implemented to be fully automatic.

Examples - Sneeds' SOFTDOC package (Sneed 1985) is a static analysis tool aimed at re-engineering and reverse engineering. SOFTDOC statically analyses source code to produce a number of tables which carry information about the control flow, data flow and interfaces of the original program. These tables can then be used as a basis for modularising and re-engineering existing COBOL programs. For examples of its use as a re-engineering tool see (Sneed and Jandrasics 1987) and (Sneed and Merey 1985).

Some work in the programmers' apprentice project has also concentrated on the development of analysis tools, particularly to analyse loops in programs (Waters 1979). The work done by Mark Weiser on program slicing also falls into this category (Weiser 1984). These approaches aim to produce information about the action a program has on specific variables.

More advanced approaches to using program analysis for reverse and re-engineering is given in (Hausler *et al.* 1990) and (Rugaber, Ornburn and LeBlanc 1990). Both of these approaches are based upon using properties of the control flow of a program to try and abstract a canonical expression for the control structures. This then forms a basis for identifying and restructuring significant sections of code.

Nearly all the systems cited in the following sections have a significant program analysis component.

2.3.2 Database repositories

Database repositories are used to store information extracted from the source code by program analysis tools. They are intended to make this information easier to retrieve and supplement, and to allow manipulation of this information making re-implementation and alterations to the code easier.

They generally store information about objects located in the source code, and relationships. Many commercial tools for reverse and re-engineering are based around this component, often abstracting objects and relationships held in an out-of-date application and allowing the migration of the software into a new environment, such as a more modern database. Thus such tools are often directly intended to facilitate software reuse as well as reverse engineering.

Examples - The two most frequently cited research projects which concentrate on this particular aspect (although in no way exclusively) are The C Information Abstractor and SRE. The C Information Abstractor (Chen and Ramamoorthy 1986; Chen *et al.* 1990) abstracts information about C programs into a database to allow analysis and manipulation of the programs. SRE is primarily interested in assisting in the maintenance and reuse of large transaction processing systems (Kozaczynski and Ning 1989)

2.3.3 User interface design

Many reverse engineering tools are based around the development of a user interface which allows the user to display and manipulate information about the code being investigated. Very simple tools such as pretty printing can be considered as being concerned with the user interface since such a tool deals with the display of already available information rather than the generation of any new information. With recent interest in program understanding and reverse engineering more complex user interfaces have been developed which allow multiple views of software (such as those generated by program analysis), code browsing (easy movement from one area of code to another) and other features typical of modern interface design.

The aim is to facilitate the understanding and manipulation of source code and provide an integrated way of displaying the results of other reverse engineering tools. Although user interface design for reverse engineering is likely to benefit greatly from general research in human computer interaction, there is clearly some way in which design should pay heed to the specific needs of program understanding and reverse engineering.

Examples - developments which are placing a large importance on the user interface component of reverse engineering tools include PUNS which has been developed to assist in the maintenance of IBM System/370 assembler (Cleveland 1989) and also Biggerstaff's design recovery tool Desire (Biggerstaff et al. 1989). MicroScope, a prototype tool to aid in the maintenance of LISP programs, is also placing a large emphasis on the development of a sophisticated user interface (Ambras *et al.* 1988).

All these tools are similar in that they provide for the display and manipulation of program information in a WIMP (Window Icon Mouse Pointer) environment. As well as displaying information, they allow additional information to be recorded and appended to the existing program representation.

2.3.4 Source code redescription tools

All the above components of reverse engineering tools have strong links to other fields of research not directly related to reverse engineering itself. Many program analysis tools are derived from work in software metrics, compiler design and program verification, while databases and user interfaces have constituted relatively major fields of interest within computer science for a long while.

In contrast, the goal of tools for redescribing source code are far more specifically related to the goals of reverse engineering. The goal of redescribing source code can be described as corresponding to the second part of Chikofsky and Cross's definition of reverse engineering, namely, the aim of redescribing source code is to "create representations of the system in another form or at a higher level of abstraction" (Chikofsky and Cross 1990, p15)

Key in this definition is the notion that redescribing source code is a *creative* process. Tools to redescribe source code attempt to 'mimic' some aspect of human behaviour that may be associated with the ability of humans to successfully perform this task, such as theorem proving or the use of heuristics. As such, redescription tools can be characterised by their use of techniques traditionally associated with artificial intelligence research.

The second important feature of this definition is that the representations created are intended to be sufficiently comprehensive to allow the user to understand the operation of the system (or part of) at the given level of abstraction. This is as opposed to views of the system which are only partial (for example a control flow graph) and need to be supplemented with other information to allow the workings of the system to be comprehensible.

The general approach taken by these systems is to use a sophisticated pattern matching technique (such as graph parsing) to attempt to match patterns from a knowledge base to a representation of source code. There is considerable difference in the nature of the knowledge base, the pattern matching algorithm used and in the goals of such systems. A more detailed discussion of the different approaches to redescribing source code will be given in the next chapter.

2.4 Software Reuse

One motivation behind software reuse has been mentioned above, namely, that many organisations have invested a lot of money and effort on software. When software becomes obsolete, this investment has to be written off. However, there are other reasons for interest in software reuse.

Software development is very capital intensive, but as nearly all software development occurs as a one off process the investment made on developing one product can rarely be used to benefit subsequent development projects. The aim of software reuse is to prevent some of this wastage by allowing some of the investment that has been made in a particular piece of software to be reused in future projects. This should improve software productivity and quality.

There are many examples of the reuse of software, such as the use of Unix programming utilities and libraries of mathematical routines (such as the NAG library). In general though the field of software reuse is primarily concerned with promoting the reuse of *software components*. The original conception of reuse through software components is usually credited to McIllroy (see Wegner 1984).

The usage of the term software component here is intended to be consistent with that of Hooper and Chester that

> "the term software component (or component) is used to mean any type of software resource that may be reused (eg. code, modules, designs, requirements specifications, domain knowledge, development experience, or documentation)." (Hooper and Chester 1991, p3)

The process of reuse through components is frequently described through the use of a "nuts and bolts" metaphor. The idea being that it should be possible to construct software using reliable, off the shelf, components in a manner analogous to the way more traditional engineering constructs artifacts out of standard building blocks.

Problems in the uptake of software reuse can be broadly divided into two areas, managerial and technical. Managerial difficulties consist of problems in encouraging people to use "off the shelf" items rather than producing all new software from scratch. Technical difficulties are concerned with allowing components to be developed, located, and linked together to form new software.

It is the technical aspects of software reuse that are of interest here. The technical obstacles to reuse mainly consist of:

1. Creating (either from scratch or through modifying existing products of software development) enough software components to make reuse viable, and defining the necessary characteristics of such components.

2. Developing mechanisms to allow components to be stored and then located when needed.

3. Ensuring that components, when located, can be suitably combined to produce new software.

The issues involved in software reuse are many and complicated and it is only the symbiotic relationship between reverse engineering and reuse that will be discussed here. It is particularly the issues surrounding the development and subsequent location of components that will be pursued in this thesis. For a fuller account of some of the problems involved in furthering software reuse, and on the next subject of domain analysis, the following should be referred to (Hooper and Chester 1991; Biggerstaff and Perlis 1989, Vol.1; Biggerstaff and Perlis 1989, Vol.2).

2.4.1 Domain analysis

The subject of domain analysis is one of the major points of convergence between research in reverse engineering and software reuse. Prieto-Diaz defines domain analysis as

"a process by which information used in developing software systems is identified, captured, and organised with the purpose of making it reusable when creating new systems" (Prieto-Diaz 1990)

This involves analysing a problem domain and describing it in terms of processes, objects and relationships.

This is very similar to systems analysis. However, whilst systems analysis is concerned with specific problems, domain analysis aims to look at a variety of systems within a given sphere of interest with the goal of abstracting generic objects and processes which characterise the domain. Once adequately described, these generic models become the basis for software reuse.

Typically the domain analysis process proceeds through a process of refinement. A model of the domain is developed through consultations with domain experts and through the analysis of documents associated with the domain. These can be considered as sources of domain knowledge. The complexity of the model developed may range from a simple taxonomy or classification scheme to fully functional models and formal domain languages.

Domain analysis can be viewed as a particular set of knowledge acquisition problems. Many of the issues encountered in developing a domain model such as knowledge representation, validation, and choice of methodology, are well established problems in the design of knowledge-based systems. Domain analysis can be seen as an attempt to 'capture' the knowledge used in systems design. This viewpoint is particularly apparent in work on The Programmers' Apprentice Project (Rich and Waters 1990) where the aim is to formalise the knowledge used by experts in constructing computer programs.

Clearly, one of the sources of domain knowledge for a domain analyst (or knowledge engineer) is the source code associated with existing systems. All of the systems discussed in the next chapter for redescribing source code are based upon the use of a domain model as a basis for their analysis of source code. The precise nature of this model can be used as a basis for discriminating between the different approaches these systems take.

2.5 Software Reuse and Reverse Engineering

Software reuse does not necessarily mean the reuse of actual code. One extreme form of reuse which demonstrates the relationship between reuse and reverse engineering is where a large, unmaintainable system is first reverse engineered to provide an adequate description of the old system, and this description is then used as the basis for the development of a new system. Such a process can be viewed as the reuse of some of the effort spent in developing the old system. Reverse engineering can be seen as necessary for the effective reuse of some of the investment that has been made in existing systems.

On a different scale, many approaches to reuse advocate the use of a repository for software components. These components can then be used as the basis for future development. One way of populating such a component library is to analyse existing systems for suitable components. For these components to be reusable they need to be adequately described, and hence this involves an element of reverse engineering. Some of the problems encountered in trying to recover reusable components from existing code as well as some possible solutions are outlined in (Basili 1988; Boldyreff and Zhang 1989; Garnett and Mariani 1990). Some other partial solutions to this problem are discussed in the next chapter and one particular approach forms the basis for the experimental work reported later in this thesis.

We have a situation whereby the reuse of much of the investment that is represented by existing systems is dependent upon reverse engineering. Similarly, later on, we shall see that the development of tools that are capable of providing a high level of support for reverse engineering, ie those that aim to redescribe systems, are to a large extent dependent upon the existence of the kind of domain models that are being developed to facilitate software reuse.

The following chapter will describe in more detail some of these approaches to redescribing source code.

Chapter 3
Redescribing Source Code

3.1 Introduction

The goals of redescribing source code equate with those of reverse engineering, to produce new documentation to improve the maintainability of software. However, whilst reverse engineering is a blanket term for a process of redocumenting a software system at many levels of detail, redescribing source code specifically aims at producing design level documentation from source code.

Reverse engineering is often described as if it could be performed as a bottom up process, that is, start from the source code and produce successively higher level descriptions of this code until a high level specification is produced. However, this is to ignore the importance of the application domain, and particularly the importance of a high level specification in providing a model of the application domain *as well as* describing the behaviour of the source code. This is what Turski and Maibaum describe as the way the specification *"binds together a program and its application"* (Turski and Maibaum 1987, p10).

The importance of the application domain in formulating a high level model of a software system means that there is a limit to the level of description we can expect to reach from source code alone without performing more analysis of the environment in which the system operates (Brown 1992).

If we are looking to semi-automate reverse engineering from source code, then we should limit ourselves to considering how to produce relatively low level descriptions of the code, and not expect to be able to automatically derive high level, abstract, descriptions of the system. Attempting to automate the process of redescribing source code therefore involves attempting to produce such relatively low level designs and specifications of code primarily from the source code. Tools that aim to (semi-)automate the process of redescribing source code can be considered as performing a form of source code analysis.

The distinguishing feature of automatic redescription as opposed to other source code analysis techniques is that the resulting description is intended to be able to *take the place* of the original source code at a higher level of abstraction. This description may be in the form of a formal specification of the code, or a description of the code's high level goals. Unlike other source code analysis techniques this derived description functions as an abstraction of the source code.

Wasserman (Wasserman 1983, p43) considers abstraction to be essential in allowing humans to manage the complexity inherent in software. Abstraction makes this complexity easier manage since it:

> "... permits one to concentrate on a problem at some level of generalisation without regard to irrelevant low level details; ..(and) to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure."

The redescription should represent source code by using terms and concepts that are closer to the application domain than those present in the source code. In this way the process of redescribing source code can be seen as an attempt to improve the link between the program and the application, the importance of which is described above.

Many source code analysis techniques that are used as tools in program understanding and reverse engineering do not attempt to automatically perform this abstraction. For example, redocumentation techniques such as those that produce cross reference information or provide details of the control and data flow of a program, do not attempt any abstraction in the above sense. In general these techniques are intended to automatically produce supplementary information about the code when current documentation is deficient and not to produce comprehensive documentation alone.

Similarly pretty printers and code restructurers also fail to qualify as redescription tools. Although such techniques produce new source code that can take the place of the old, they fail to perform any significant abstraction of the original code.

3.2 Approaches to Redescribing Source Code

Current research into redescribing source code can be divided into three according to the general approach that they take. *Transformational* approaches are based on mathematical notions of program equivalence. *Plan based* approaches are based upon a psychological theory of programming founded on the notion of a programming plan. *Reuse based* approaches are based around a domain analysis of the relevant area of software design for the purposes of software maintenance and reuse.

3.2.1 Transformational approaches

There has been considerable interest for some time in approaches to software development based upon the idea of stepwise refinement as first described by (Wirth 1971). This is a software development methodology which advocates moving from a high level specification to an implementation through a series of steps in which the description of the planned system is transformed successively to a more concrete version (ie nearer to implementation). The big advantage of developing systems by using stepwise refinement is that each individual step can be rigorously checked to ensure that errors are not introduced into the final system through the development process.

One common approach to software reuse is to develop a library of abstract programs that can be automatically transformed into more concrete forms through the application of a series of program transformations (see Biggerstaff and Perlis 1989, Vol.1, pp321-413). Transformational approaches to redescribing source code aim to reverse this process, and to use program transformations to transform existing into a more abstract representation.

There have been two main attempts to use program transformations to redescribe source code. One attempt has been developed by Ward as part of work on the Maintainers Assistant (Ward 1989; Ward, Callis and Munro 1989). The other has been developed by Lano and Breuer as part of the REDO (Restructuring, Maintenance and Validation of Software Systems) project (Lano and Breuer 1990). There has also been a less ambitious approach to using program transformations in a reverse engineering context as part of the Practitioner project (Boldyreff et al. 1990; Boldyreff and Zhang 1989).

The technology of the above tools is very similar to that used in tools for formal program verification such as SPADE (Carre and Clutterbuck 1988). The use to which these tools are put are distinct. Program verification tools are intended to verify that code conforms to a *preexisting* specification, whilst transformational tools for redescribing source code aim to produce a specification by applying transformations to the original code.

The general transformational approach is depicted in *figure 3.1*. The first stage converts source code into an intermediate language. This is a straightforward translation aimed to allow such systems to deal with code written using different source languages. This intermediate language is typically small and well defined and this allows properties of the transformations to be checked easily.

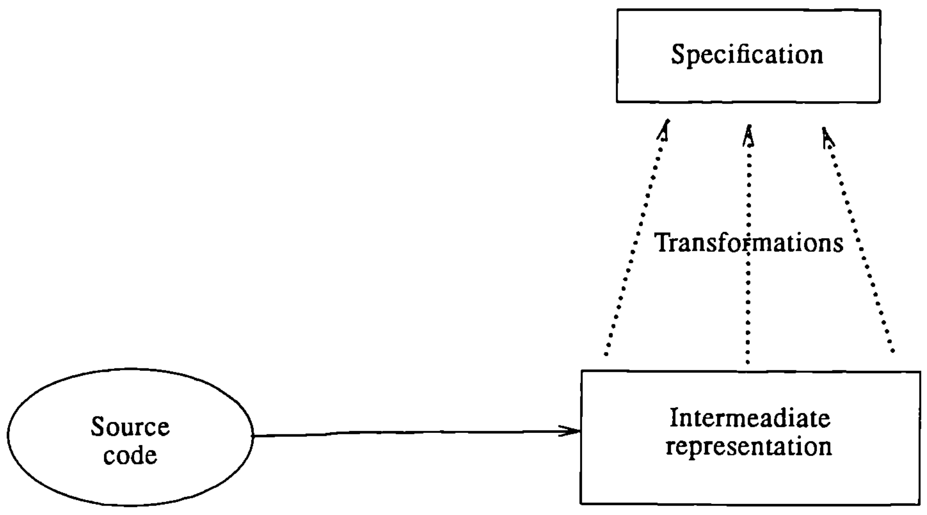


Figure 3.1: The generalised transformational approach

Once this intermediate representation has been generated, transformations can be selected and applied to the program. The choice of which transformation to apply is made either automatically or with user direction. The aim of these transformations is to successively transform the description of the source code from a low, implementation, level to a high level mathematical description that is functionally equivalent to the original code. This transformed program is then intended to function as a specification for the source code.

The transformations used by these systems are intended so to be *correctness preserving*. In this way the derived specification is shown to be correctly implemented by the code.

The approaches of Ward (Ward 1989; Ward, Callis and Munro 1989) and of Lano and Breuer (Lano and Breuer 1990). are very similar. The differences between the two stem from the theoretical basis they use for developing their theories of program transformation. Ward bases his theory on an infinitary logic language whilst Lano and Breuer's work uses a combination of category theory and the theory of monads as their basis for developing similar theories. It is unclear as to what practical difference this may make in the implementation of such theories, but it seems unlikely that there would be any major divergence in the performance of two otherwise similar systems.

The approach of Ward has been implemented as part of a tool for reverse engineering and software maintenance, "The Maintainers Assistant". This tool aims to assist in the maintenance of assembler code from IBM's CICS product. It is unclear how much of Lano and Breuer's work has been incorporated into a particular tool.

The use of program transformations in the Practitioner project has been far less ambitious (Boldyreff and Zhang 1989). Program transformations are used to transform programs into equivalent recursive versions. In this way pieces of code can be decomposed into a collection of recursive procedures that, with the addition of manually supplied comments, function as a higher level description of the original code. In practice this approach is more of a code redocumentation tool, in the sense of pretty printers and code restructurers, rather than an attempt to automatically redescribe code at a higher level of abstraction. The transformational component of the tool is fairly straightforward and the increase in the level of abstraction of the code is relatively low.

One of the main advantages of transformational approaches in general is their ability to deal with *any* source code. We will see later that recognition based approaches are unable to deal with some source code and this limits their applicability.

The disadvantage of the transformation based approach mainly stems from the difficulty involved in manipulating programs as mathematical objects. This is a difficult and time consuming task as the results of work in program verification has shown. Attempting to obtain formally correct specifications from existing code is likely to present the same problems of complexity. There is also the danger of producing specifications that are difficult to understand since the quality of a specification does not rest solely on its formal correctness but also on its ability to link the code to the application domain.

3.2.2 Plan-based approaches

These approaches are all based on a psychological theory of programming skill centered around the notion of a programming plan. Programming plans are posited as a major way in which experienced programmers organise their knowledge about programming. Their use as a way of explaining programming expertise has been explored by a number of different people, most notably Elliot Soloway and others at Yale University (Soloway et al. 1982; Soloway and Ehrlich 1984a; Soloway and Ehrlich 1984b) but also in a slightly different form by work on the Programmers Apprentice at MIT (Rich 1981; Rich and Waters 1990), and others (Gilmore and Green 1988; Rist 1986).

Although the precise nature of programming plans varies, the essential idea remains relatively constant. A programming plan encodes knowledge about how to implement a particular goal. This goal may be high level, such as "implement a payroll program", or much nearer to implementation such as "swap values of variables".

These plans are intended to correspond to the fundamental and language independent structures used by expert programmers to organise knowledge of programs and programming. Examples of such programming plans are *list-length*, *calculate-average* and *linear search*.

Plan-based approaches to redescribing source code can all be seen as attempting to automate some aspect of "program understanding". Whilst this is not explicitly equivalent to attempting to redescribe source code, all these systems consider a major motivation of such automation as the production of new documentation for existing code. Thus they can correctly be considered as attempts to redescribe source code.

The basis of all plan-based approaches to redescribing code is perhaps best summed up by the view of Letovsky, that to understand a program "means knowing the entire goal hierarchy" (Letovsky 1988, p9). The high level nodes of this hierarchy then correspond to the goals of the program while lower level nodes correspond to the way that a particular goal has been implemented.

Currently there have been four approaches to redescribing source code that fit into this category. At MIT there has been the development of Wills' Recogniser (Wills 1986; Wills and Rich 1990; Wills 1990), and the work of Letovsky in developing a system called CPU (Letovsky 1988). These developments have been part of the Programmers Apprentice project which aims to develop a system to act as an 'intelligent' assistant to software engineers (Rich and Waters 1988; Rich and Waters 1990).

At the University of Illinois, Jim. Q. Ning (Ning 1989; Harandi and Ning 1990) has developed a system called PAT (Program Analysis Tool). The most recent approach to plan based analysis of programs is Hartman's UNPROG system developed at the University of Texas at Austin (Hartman 1991).

It should be noted that there are a number of other systems developed for the automatic debugging of programs that are related to the systems described here, for example PROUST (Johnson and Soloway 1984), PELAS (Korel 1988), LAURA (Adam, A. Laurent and J-P. Laurent 1980) and also (Wertz 1987). However, these systems do not attempt to generate new descriptions of the source code they are given, and many of them work by comparing the programs they are given with model programs. For these reasons they are not considered here.

The first three of these systems; The Recogniser, CPU, and PAT, are very similar in their general approach. They aim to demonstrate the feasibility of their approaches on small student programs before attempting to scale the systems up to deal with larger and more realistic examples. Thus they aimed for detailed analysis of simple programs, this analysis forming the basis for a computerised "understanding" of the program. The general approach taken by Will's, Letovsky and Ning is illustrated in *figure 3.2.*

These three systems all convert a library of plans (or clichés as Wills refers to them) into a knowledge base which contains the information required for identifying these plans in source code. Input source code is translated into an intermediate form which is intended to promote language independence, and to reduce the amount of variation in the way a particular plan can be implemented in source code. This aims to improve the efficiency of the matching algorithm which attempts to match plans from the knowledge base onto this intermediate form. A comparison of the different representations and matching algorithms used is given in *table 3.1.*

Wills' recogniser (Wills 1986; Wills and Rich 1990; Wills 1990), first transforms LISP source code into a flow graph representation, where a flow graph is a labelled, directed, acyclic graph (Wills 1990, p 125). This is done through an analysis of the control and data flow of the original source code. Plans, or clichés as they are referred to in this work, have to be translated

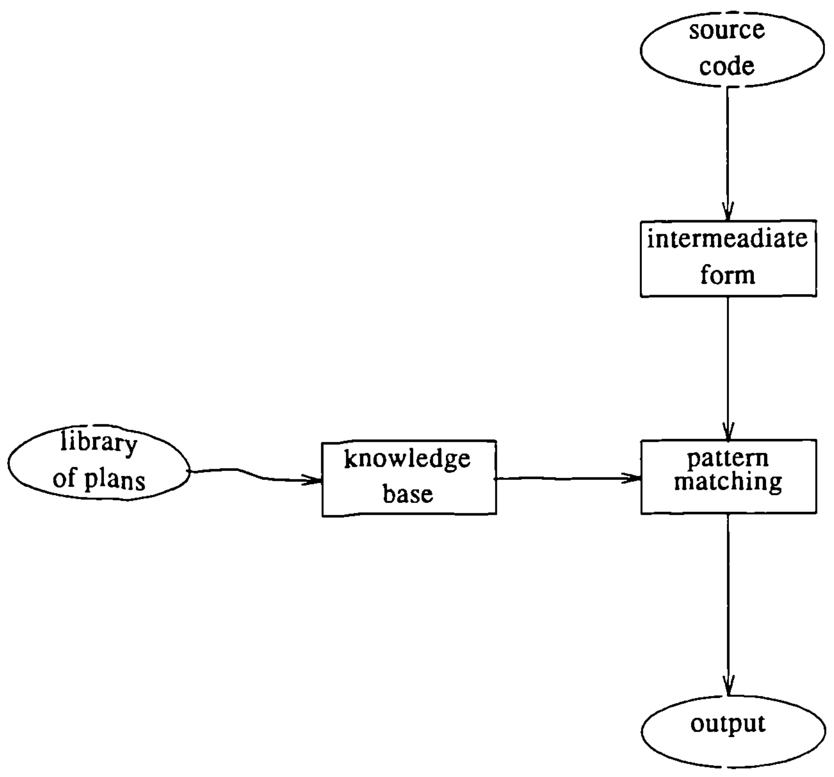


Figure 3.2: The generalised plan based approach

(currently manually) from the plan calculus form on which the Programmers Apprentice work is based to generate a graph grammar.

A graph parsing algorithm based on an algorithm developed by Brotsky at MIT (Brotsky 1984) is then used to parse the transformed source code according to the rules of this grammar and generating a goal hierarchy for the program. Once a successful parse has been achieved, natural language documentation for the recognised code is produced. Even if a fully successful parse is not achieved, some clichés can still be identified in code as part of a partial parse of the program flow graph.

Letovsky's CPU (Letovsky 1988), also developed as part of the programmers' apprentice project, is in practice very similar to the

Author	System	Knowledge Base	Intermediate Representation	Matching Algorithm
Wills	Recogniser	Graph-grammar	Flow graph	Graph parsing
Letovsky	CPU	Transformations	Lambda calculus	Canonicalisation
Ning	PAT	Inference rules	Program events	Inference

Table 3.1: A comparison of different plan based approaches

transformational systems described earlier. Its analysis is based on plans that are represented by correctness preserving transformations between lambda calculus expressions. Source code is translated into lambda calculus expressions and then these transformations are applied to rewrite the code in a semantically equivalent form. The aim is to reduce the representation of plans and of the source code to a canonical form so that the problem of pattern matching becomes trivial. In practice only a limited degree of canonicalisation, termed by Letovsky "quasi-canonicalisation", is achieved. This is still sufficient to allow some recognition to occur although CPU does suffer from considerable performance problems, being unable to successfully analyse the 300 line FORTRAN program it was specifically designed for.

The reasons for considering CPU as a plan based system rather than a transformational system are twofold. Firstly, the transformations are explicitly derived from programming plans rather than from mathematical theories of equivalence (in fact it is not clear on what basis Letovsky makes his claim that derived descriptions are semantically equivalent). Secondly, unlike transformational approaches, the intermediate steps in the transformation process are considered to be 'meaningful' in that they constitute the goal hierarchy represented by the source code.

The approach used by Ning (Ning 1989; Harandi and Ning 1990) in the development of PAT is very similar to that taken by Wills' Recogniser, although the actual realisation of the mechanism is very different. PAT represents source code as a series of program events. Program events are organised as a hierarchy, with the lowest level corresponding to simple programming language structures, whilst the higher level events correspond to more plan like structures. These higher level structures are inferred to be present in the source code by the analysis PAT performs.

Program plans are represented as inference rules. Analysis proceeds through the repeated application of these pattern-directed inference rules to derive new program events from existing ones. A truth maintenance system is used to monitor and control this application of rules. In this way a goal tree for the program is constructed.

The most significant difference between PAT and The Recogniser is that PAT encodes knowledge about common errors made when implementing plans. This enables PAT to identify errors in code by recognising incorrect instances of a particular plan. This increases the range of code that PAT can identify.

Hartman's approach (Hartman 1991) is slightly different in that he considers that the hierarchical analysis as performed by the three systems outlined above is too detailed to allow such systems to operate successfully on commercial source code. Thus Hartman has attempted a less ambitious analysis of code with the intention of being able to apply and verify his approach on commercial programs.

He has restricted himself to trying to identify simple plan like structures within programs. This is done by representing source code as a hierarchical control flow/data flow graph. This graph is then decomposed into primes which reduces the complexity of the analysis. Flow graph primes are to flow graphs what prime numbers are to the natural number system. Any flow graph can be decomposed into a prime decomposition which represents the original graph as a combination of primes. Moreover, as with prime numbers, the decomposition of any particular graph is unique and so a *proper decomposition* of the original flow graph can be produced. (This process is similar to the decomposition used by Fenton and Kaposi (Fenton and Kaposi 1987) to produce software structure metrics.)

Plans are identified in the original code by matching flow graph representations of them against this proper decomposition of the program flow graph. The results of this work suggest that such an approach is generally applicable and may lead to useful applications in redocumenting and restructuring source code. However, the plans that UNIPROG currently identifies in source code are at a very low level, corresponding only to very simple control structures. It will probably be necessary for the system to be able to identify higher level structures before this kind of approach finds useful application.

3.2.3 Reuse based approaches

In contrast to plan based approaches which are based upon psychological theories of programming skill, reuse based approaches are based on a domain model of the relevant area of expertise. In contrast to plans, the structures searched for in source code tend to correspond to higher level, application knowledge rather than low level programming structures. The two reuse based approaches to be described are that of Biggerstaff (Biggerstaff 1989; Biggerstaff et al. 1989) and the far smaller project carried out by Karakostas (Karakostas 1991).

These approaches are characterised by their use of an expectation-based approach to identifying design and application concepts in source code, and also by the relative informality of their approach. That is, they place far less emphasis on obtaining formally correct descriptions of the source code than the transformational and plan based approaches described so far.

Karakostas' system IRENE (Karakostas 1991) aims to identify concepts from the application domain in source code by using a combination of knowledge about relationships between concepts and prototypical implementations of such concepts.

IRENE analyses COBOL code that has been "reverse parsed" into an intermediate frame like language. Analysis knowledge is encoded in frames which describe prototypical features of implementations of particular concepts. The analysis of various hypotheses about concepts present in the original code involves comparing features of the code with the prototypes. This leads to calculating a degree of plausibility for the hypothesis based upon the weights associated with particular pieces of evidence. At the time of writing many intended features of IRENE have yet to be implemented.

Biggerstaff's project (Biggerstaff 1989; Biggerstaff et al. 1989) aims to assist in what Biggerstaff terms "design recovery". Design recovery aims to produce a detailed and multi-faceted description of the design of existing software. Central to this is the goal of identifying "conceptual abstractions" in code. To this goal, two related systems are being developed, DESIRE and TAO.

DESIRE is based upon the development of a rich domain model in the form of a semantic net. This is used as the basis for an analysis of source code based on both formal and non-formal features. DESIRE uses this domain model provide the information to perform an expectation driven search of source code for "conceptual abstractions" which are used to represent application specific knowledge of software design.

It is intended to use the domain model initially to produce high level expectations about the conceptual abstractions that may be present in the code. This initial search is to be mainly for "linguistic idioms", lexical patterns that indicate the occurrence of specific conceptual abstractions. Once expectations have been set up the semantic net is used to perform a more detailed analysis of the code based upon the fine grained structures and relations that may be present. In this way a detailed model of the code in terms of the design abstractions that it implements is built up.

TAO is intended to help in the search for conceptual abstraction by utilising a *connectionist* approach to source code analysis (see Rumelhart and McClelland 1986 for an overview of approaches to connectionism). A richly connected network of nodes and links are to provide a distributed representation of the conceptual abstractions contained in the domain model. Knowledge is encoded in the connections between nodes and the weights associated with these connections.

The hope for such an approach is that by representing domain knowledge in a connectionist network it should be possible to integrate many different sources of information into the analysis. Connectionist networks are also able to learn from experience by adjusting the weights associated with the links between nodes when given examples. It is possible that a system like TAO could learn domain knowledge in this way through being given examples of source code, although there are many difficulties associated with learning in this fashion.

At the time of writing, most of Biggerstaff's work (as published) is at an early stage of development, with development concentrating on the domain model and user interface. The rest of the work, and in particular the work on TAO, is at a very early stage and so it is difficult to comment on it in detail.

3.3 Summary

This section has characterised the notion of redescribing source code and classified current approaches into three categories.

- *Transformational* - These are based on mathematical theories of program equivalence. They aim to derive formal specifications from source code. These approaches place great importance on the code being a formally correct implementation of the derived specification.

- *Plan Based* - These are based on psychological theories of programming skill, centered around the notion of a programming plan as a fundamental mechanism used for structuring programming knowledge. They aim to produce a hierarchy of plans present in source code, and to use this as a basis for automatic program understanding. They place varying degrees of importance on the correctness of their output.

- *Reuse Based* - These are based on the development of an application specific domain model of areas of software design. They aim to extract information from source code in the form of design and application level concepts embodied in the code. Being able to extract useful information from code is considered to be more important (initially) than the formal correctness of such information.

Chapter 4
Source Code as Text

4.1 Introduction

The aim of this chapter is to emphasise the textual nature of source code and to describe the role that non-formal information in source code plays in the interpretation of such a text. This viewpoint is used to provide a critique of the research described in the previous chapter, and suggest some possible directions for research into redescribing source code.

Firstly, the distinction between formal and non-formal information in source code is described. This distinction is used to characterise the formalist position regarding the meaning of source code which appears to have been adopted by many of the researchers involved in work to redescribe source code.

The difficulties of such a formalist position are illustrated, and a view of the meaning of source code as being generated by its interpretation as a text developed. Applied to the work described in the previous chapter this orientation provides a critique, the main conclusion of which is to suggest that approaches to redescribing source code should pay more attention to the use of non-formal information in their code analysis.

The final part of this chapter then attempts to suggest ways in which a view of source code as text could lead to useful methodologies and tools for source code analysis, one such approach being the basis of the experimental work reported in this thesis.

4.2 Formal and Non-formal Information

Biggerstaff in his paper on design recovery (Biggerstaff 1989) makes a distinction between formal and informal information in source code. He fails to fully explicate this distinction but illustrates it by way of an example using three different versions of a program written in C. These examples constitute a movement from source code containing much informal information to an equivalent piece of source code with no informal information included. The two extremes of these examples are equivalent to the examples of Pascal source code in *figure 4.1* and *figure 4.2.*

```
{The following procedure writes a line of text to the
 standard output and terminates with a new line     }

PROCEDURE writeline(line:lines)

  VAR i:0..linemax:

  BEGIN

   i := 1;
   while line[i] <> slash do begin   {slash marks end of line}
    write(line[i]);
    i := i+1;
   end;
   writeln;
  END; {writeline}
```

Figure 4.1: Pascal program with informal information

Figure 4.1 has comments and meaningful identifier names, such as "line" and "slash". These are what Biggerstaff terms informal information as these do not relate directly to the operation of the code but relate to its ‘readability’. In *figure 4.2* this informal information has been removed to leave code which is very difficult to understand. Biggerstaff notes that whilst the complier will treat these two programs equivalently, for the version with no informal information (ie *figure 4.2*) "Interpretation and understanding of the program has become impossible in any deep sense" (Biggerstaff 1989, p41).

```
PROCEDURE #001(#002:#003)

  VAR #004:0..#005:

  BEGIN

   #004 := 1;
   while #002[#004] <> #006 do begin
     write(#002[#004]);
     #004 := #004+1;
   end;
   writeln;
  END;
```

Figure 4.2: Pascal program with informal information removed

Biggerstaff, however, has not gone far enough in removing all the features of the text that are unnecessary to produce compiled code. His second version of the program still has a neat layout, this is not strictly necessary for compilation. Also the names of the programming language constructs (eg "LOOP", ":=", "+", etc.) are not entirely arbitrary but appeal to an intuitive grasp of their intended operation. A compiler could be designed to compile code where each of the constructs of the language are given meaningless names in the same way that identifiers have been given such names in *figure 4.2*. The net result of such a translation would be source code in the form of a string of seemingly arbitrary terms. One possible example of such source code for the example program is given in *figure 4.3*.

Figure 4.3 can be considered the "compilers eye view" of the source code of *figure 4.1*. *Figure 4.3* is clearly even less informative about the code's function than *figure 4.2*. Yet the organisation of *figure 4.3* still has enough structure (in fact the same structure) that allowed the first two examples to be

```
#007 #001 a #002 10 #003 s #008 #004 10 0   #005 cc 9 #004 20 144
cc %l #002 d #004 f + #006 ˜ e a #002 d #004 f s cc #004 20 #004
k 1 cc ** cc  cc ** cc
```

Figure 4.3: Pascal program with non-formal information removed

compiled into equivalent code. Biggerstaff appears to have been too conservative in his notion of informal information. There are clearly many features of ordinary source code that are not included in Biggerstaff's usage of the term *informal* that are used to improve readability but are not strictly necessary to define the eventual action of the code.

I wish to revise Biggerstaff's distinction to take into account the above example. The distinction I believe Biggerstaff was intending to make was between those features of the source code that are semiotic in nature, that is those that are constitutive of the system of codes that allow source code to be interpreted as a text, and those that form part of the closed system of the text. The former will be termed *non-formal information* to distinguish between this definition given here and Biggerstaff's *informal* information. This distinction is necessary since Biggerstaff's usage of informal does not include such features as the layout of the code or the semiotic role of the program language statements.

The label *formal information* will continue to be used to refer to those aspects of the internal structure of source code that relate to the nature of the compiled code, and more specifically to those aspects that are used in the construction of what will later be termed a *formal program model.*

To fully explicate this position I wish to use a general semiotic approach as a basis for discussing the approaches to redescribing source code outlined in the previous chapter. Semiotics aims to study the nature of sign systems and signification in general, as such it encompasses the study of language and text. In the first half of this century, the philosopher C.W. Morris distinguished three branches of semiotic enquiry; *syntax, semantics* and *pragmatics*. Morris defined syntax as the study of the relations of signs to one another, semantics as the study of the relation between signs and the objects to which they are applicable, and pragmatics as the study of the relationship of signs to the users of the signs. (This account is derived from Levinson 1983, p1).

A precise distinction between these particular branches of study has proved very difficult to obtain, however the study of semantics has increasingly become identified with the notion of 'meaning'. Part of the argument expanded here is that in certain fields of computer science the role of pragmatics in generating meaning in a broad sense has been overly neglected.

It is possible to view source code and design descriptions as texts and hence as complex signs. Such texts can then be analysed from a semiotic perspective, that is from their ability to stand for something other than themselves and so be used as *representational devices* (Tippets 1988). From this semiotic orientation, two main criticisms will be made of current approaches to redescribing source code. It will be claimed that most approaches make the following questionable assumptions:

1. That the 'object' described by source code is primarily a *formal program model*, that is a mathematical model of the anticipated operation of the source code.

2. That the pragmatic properties of the original source code and the derived design description can be neglected in favour of considering primarily the correspondence between the formal semantics of the design description and the formal program model.

In fact, both these assumptions are based upon viewing the relationship between an object and a description of that object as straightforward, ignoring the problems raised by the nature of representational devices such as texts and diagrams etc. These assumptions can broadly be described as *formalist* (see Leith 1990, chapter 2, for a fuller description of this term) since they are both based upon considering the 'meaning' of some text to be some formal and decontextualised expression of the text's semantics.

4.3 Difficulties with a Formalist Approach

To show the complex nature of the relationship between some textual description of an object and the object itself will involve examining the relationship between some formal object and its associated description. This necessitates defining some of the notions that will be used later on, particularly the notions of *formal system*, *formal structure* and *formal equivalence.*

4.3.1 Formal systems

A formal system consists of a *syntax* and a relation of *derivability*. The syntax, which is defined over an alphabet of symbols, defines the sentences or well formed formulae (wff's) which constitute the formal language of the system. This formal syntax enables sentences to be parsed as wff's. (Once parsed these sentences can be given a semantics, this semantic form defines the wff's relation to the rest of the formal system.)

The relation of *derivability* is a relation between the wff's of the system such that for any wff to be a *theorem* of the system then it is either:

1. An *axiom* of the system (that is a member of a given set of wff's)

2. Derivable from the *axioms* of the system

So, for example, this derivability relation can be used to *prove* that a particular sentence in the predicate calculus is a logical consequent of some other sentence, or similarly prove that a formal model of a piece of source code possesses a particular property. In both these cases the formal system is being used to manipulate meaningless (to the system) symbols in the same way that computers do.

The importance of formal systems to computer science is in the connection between formal systems and mechanical operation since we can define a mechanical, and hence automatable, operation directly (eg in terms of a Turing Machine) and then define a formal system as one whose set of theorems can be generated by such a machine. Conversely, we can define a formal system directly and then define a mechanical operation as one that is computable in some formal system (Smullyan 1961, p1). Given this we can see computers as machines for automating formal systems. This is the basis of The Turing Thesis that for any deterministic formal system there exists an equivalent Turing Machine (Haugeland 1985). Thus for any process to be automatable necessarily entails that the 'rules' of the process can be captured as part of a formal system.

Within a particular mathematical framework it is possible to define notions of formal equivalence. These can be used to determine when two expressions in a particular system can be considered the same, or when two formal systems themselves can be considered equivalent. This notion of formal equivalence is very important for all mathematics and for formal systems in particular. It is this ability to re-express a statement in a mathematically equivalent form that is central to many approaches to redescribing source code.

The notion of a *formal structure* aims to capture what is common to equivalent representations such as the two example programs of *figure 4.1* and *figure 4.2*. Any text written using a formal language, that is a language which has been formally defined so that expressions of the language are able to form part of a formal system, can be considered as defining a formal structure *with respect to* a particular notion of equivalence. This is how two different examples of source code can be said to define the same 'program', and how two different predicates in a logic language can pick out the same property.

There are many different notions of equivalence that can be imposed upon a formal system. Any particular mathematical definition of equivalence can be considered as providing an aspect from which to view and compare two systems or expressions. This view defines the formal structure of the object. However, the particular notion of equivalence used is often chosen to accord with some intuitively held notion, such as program or logical equivalence, which exists only as an informal ideal. Whilst this formal definition of an intuitive notion provides a way of mechanising tests for equivalence (and other properties of formal structures), this formal definition usually fails to capture all the properties entailed by the original ideal.

One of the arguments to be developed here is that the notion that source code is about "something" (ie about the operation of a computer) has been mathematicised in this way by considering that source code defines a formal object. Unfortunately this often leads to an overly simple view of source code as fully equivalent to a particular formal object often termed a "program".

4.3.2 Interpretation of a formal structure

By definition any particular formal structure, regardless of what this structure is intended to represent, can be presented in many different ways. Given a suitable formal definition, all these representations can be considered as equivalent. However, from a different perspective these equivalent representations can have very different properties.

When a text is considered as a representation or description of some object of interest, it is precisely the relation between the text and the object *as perceived* by the reader that determines the accuracy or correctness of the description. This relationship depends upon the power of the text to stand in place of the object that is being described, and not directly on the mathematical properties assigned to the text through the definition of some formal structure. Often though, the formalist perspective overlooks the importance of this pragmatic relationship, preferring to view the ability of a text or formal system to represent an object in terms of the semantic relationship between the expressions of the text and the object of interest.

The relationship of description to object can be considered as analogous to that of a map to the land that it represents, this relationship perhaps being the clearest example of the correspondence between an object and its description. The limitations that can be seen to apply to the map as a representational device can also be seen to apply to all other types of representation including texts.

The map does not simply replicate the land that it covers as a life size model. If it did it would be 'equivalent' to the land in every way and so would cease to have any value as a 'map' as distinct from the territory itself. What a map does is to leave out many of the features of the land in favour of highlighting those that are useful to travellers, and also by including additional information that could not be easily be gained from inspecting the land directly, such as contours lines. In short, it is important to remain aware that "The map is not the territory" (Korsybski 1958)

The description of the land provided by a map is always an abstraction away from the 'reality' that it seeks to describe. This is true of any representational device. Even the most rigorously defined formal representations, are unable to represent the object of interest in some 'neutral' or 'objective' way. But instead can only represent an object from a particular viewpoint.

The objections to a view of mathematical descriptions as objective that will are highlighted are two fold. Firstly, the correspondence between any formal language expression and the object that such an expression is referring to cannot be simply defined within a mathematical system. Secondly, providing a definition of a formal structure is not the same as providing a useful description of that structure.

The first objection stems from the problem of defining the correspondence between an expression and the object that it represents. Consider the following expression in some formal logic:

All men are mortal

This expression has two roles, firstly it can be considered as part of a formal system that implements logical inferences. Thus such a statement can be used within a formal system to produce logically correct inferences, ie given some other axioms, it might conclude that,

Socrates is mortal

The symbolic manipulation used to produce the above conclusion occurs entirely within the formal system and so can be mechanised as part of a system of logic. But the first expression also has a second role. As well as forming part of a closed system, it is also intended to be interpreted as representing some 'fact' about the world. It is intended to correspond to some empirical knowledge. As such the expression represents some state of affairs, in this case that all members of the human race only live for a finite period of time.

The accuracy of this correspondence between expression and the state of affairs that it represents is always dependent upon interpretation. The interpretation of the term "men" in the above example depends upon the conventional usage of the term. It is this usage which enables us to interpret what "men" is intended to signify. The apparent transparency of logic as a representational language is deceptive, since before a transparent expression can be arrived at, the definition of the terms to be used in the expression must be resolved. Thus the ambiguity of the term "men" leads to doubt as to whether the expression is intended to refer to the whole of the human race or only to the adult males of the species.

Often some form of truth conditional semantics is suggested to allow the meaning of such terms to be unambiguously defined. However such a definition still rests upon the interpretation applied to other natural language terms and so on. Any form of truth-conditional or model-theoretic semantics is based upon the existence of an objective world containing given, ready made objects and concepts. Linguistic expressions function as 'pointers' to this objective universe with the truth of an expression being based around some notion of correspondence between the expression and this universe.

If the Sapir-Whorf hypothesis that language influences the way we perceive the universe is accepted (Whorf 1956), then the existence of an independant and objective universe becomes an over-simplification of the actual situation. In many cases, where there is considerable agreement and stability in the conventional interpretation given to terms this idea of an objective universe is an adequate approximation to the real state of affairs. This explains the success of formalisms in fields such as the natural sciences and engineering. However, even in these fields the construction of this stable consensus which allows the application of formalisms has occurred over time.

To expand upon the above point, even when the object that is being described is formally defined (as in the case of source code) we still need to consider the distiction between *definition* and *description.* The formalism defines the object, but it is the role of the representational device, usually a text, to describe the object. This involves not only considering notions of correspondence between representation and objects, but also considering the pragmatic aspects of the use of the representation in providing a channel for communication.

Let us take an example from mathematics to illustrate this distinction. Hilbert's program of axiomatisation in mathematics has led to the structure of whole fields of mathematics being defined in terms of a small number of axioms. Hilbert‘s aim was to provide an epistemic foundation for mathematical knowledge by providing objective definitions of basic mathematical concepts such as 'theoremhood' and 'proof' based upon the manipulation of meaningless symbols within a formal system.

One field of mathematics that was successfully axiomatised was that of Number Theory. A set of axioms are used to define a formal system which corresponds to the structure of the natural numbers. In this way, any expression in the formalism of number theory is a theorem if and only if it is derivable from the original axioms. What is required for an expression to be a theorem is rigidly defined.

This formulation of the natural numbers defines the formal structure of the natural number system, however, it does not *describe* this structure. Describing this structure, and identifying which theorems are of interest and which are not is the role of mathematics and mathematicians. In this sense, the role of the mathematician is in finding an adequate description for a complex structure. This role is no different to that of any one who aims to produce a description of some object of interest, such as a scientist or knowledge engineer. For a description to be adequate it must accord with the conventions that are commonly used by other mathematicians so that it can be comprehended by others. In this way, mathematical discovery can be viewed as the development and acceptance of such conventions.

This view of the development of mathematical concepts is well illustrated by the work of Imre Lakatos (Lakatos 1976). His work into the nature of mathematics, and particularly the nature of proof, has demonstrated the major role of social processes in mathematical discovery and in the the definition of mathematical notions.

For similar reasons Wittgenstein has argued against the the view of mathematics as having some privileged access to truth, preferring to see mathematics and logic as specialised language games within the larger sphere of natural language. This view of mathematics is perhaps well summed up by Wittgenstein when he states that

> "The mathematician is an inventor, not a discoverer." (Wittgenstein 1978, I-168).

A system written by Doug Lenat (Lenat 1982) called AM demonstrates the importance of this distinction between definition and description in a practical way through an artificial intelligence attempt to mimic the process of "discovering" mathematical concepts. AM was provided with some heuristics for producing such concepts from older concepts within the domain of number theory. An example of such a concept would be that of prime numbers (a concept that AM did manage to "discover"). After about an hour of run time, after which the rate at which AM discovers new concepts slows significantly, the system had discovered about 300 such concepts. Of these only about 25 where deemed to be mathematically interesting, whilst about 175 were considered to be worthless.

Similar problems to those encountered by AM are likely to become manifest in attempts to use the formal structure of source code as the sole basis for redescription tools. To adequately describe source code it is necessary to use expressions that are defined by consensus. Unfortunately, the establishment of this consensus is not a formal process but a social one. This point is particularly important when dealing with expressions that represent application domain concepts. To obtain definitions of these objects in a form that enables them to be recognised within source code it is necessary to step outside the world of formal objects and into the application domain itself.

Even when there is a reasonably stable consensus concerning the formal definition of objects, such as that of certain abstract data types, there is still the problem of finding a representation for these objects which allows them to be identified within code. Only in very special cases will a formal definition of an object be sufficient to define the conditions of appropriateness which determine when that definition can be applied. This is much like natural language parsing where a dictionary style definition for a term is far to simplistic a representation of the 'meaning' of that term to allow parsing on any reasonable scale to be practical.

To try to overcome some of the problems associated with a formalist viewpoint, as described above, the next section considers work from fields associated with describing how humans are able to interpret and communicate. This involves considering work within the general fields of hermeneutics, semiotics and linguistics.

4.4 Texts and Textual Interpretation

Having discussed some of the problems involved with the correspondence between descriptions and the objects that they represent. It is necessary to discuss an approach to the problem of how a particular representational device, namely text, is able to provide a medium for communication. In the main this section will center around the semiotic theories of Umberto Eco although it will also be necessary to introduce some of the ideas characteristic of the hermeneutics of Heidegger, Gadamer and Ricoeur. (A prior application of some of these ideas to computer science, though primarily oriented towards artificial intelligence research, can be found in (Winograd and Flores 1986).)

What are the characteristics of a text? Paul Ricoeur defines a text to be "any discourse fixed by writing" (Ricoeur 1981, p145). This is a very general definition, and clearly includes source code and design level descriptions since these are used as a means of conveying information to other people. Such a definition concentrates on the role of the text as a medium for communication. We can view a text as being produced by someone, the author, with the purpose of communicating a message to the intended readers of the text. How does the organisation of a text provide for such communication.

Eco considers a text as being *multilevelled* (Eco 1976, p57-58). A text does not express a single content or denotation but conveys many interrelated messages. Through these interelated messages a text is able to describe or create a possible world. This world is appropriated by the reader through interpretation. Such a world can be imaginary as with novels, or refer to objects present in the empirical world. The distinction between real and imaginary is not important here since the mechanisms involved in interpreting texts are identical regardless of the nature of the object or objects they seek to describe.

To be intelligible a text has to use symbols and codes that are defined by convention to convey its message. This is a feature of texts regardless of the nature of the object being described.

> "To organise a text, its author has to rely upon a series of codes that assign given contents to the expressions he uses. To make his text communicative, the author has to assume that the ensemble of codes he relies upon is the same as that shared by his possible reader." (Eco 1979, p7).

It is only through such shared codes that communication is possible. These codes govern the correlation between an expression and its content and so such codes are responsible for the way linguistic expressions are interpreted.

Biggerstaff talks about source code having an "informal semantics" as well as a formal denotation. However, the use of the word "informal" leaves it vague as to what such semantics are referring to and suggests that such semantics are somehow arbitrary. This is another reason for preferring to use the term non-formal here.

What is the nature of these non-formal semantics? Eco's semiotic approach considers that:

> "The semiotic object of a semantics is the *content*, not the referent, and the content has to be defined as a *cultural unit*" (Eco 1976, p62, italics in original)

A cultural unit is anything that is defined or distinguished as an entity by a culture. A cultural unit acquires its meaning from its position within a *semantic field.* So for example the term <dog> does not denote some physical object or objects, or some set such as that which contains all possible dogs. The meaning of the term is an abstract entity which is defined by cultural convention.

This definition of a cultural unit, based upon the communicative role of texts, frees us from having to make distinctions between texts and terms that refer to abstract or imaginary objects and those that attempt to represent some state of the empirical world. Note, however, that to state that a cultural unit is defined by convention is *not* to say that such a definition is necessarily arbitrary (in fact such definitions are very rarely truly arbitrary).

A good example from computer science of the way a semantic field becomes delineated and described through language is the field of data structures. There are an infinite number of different data structures that could be used in design and programming, but this field has been sliced up into discrete units, such as those that correspond to the cultural units; <stack>, <queue>, and <linked list>. These are used to represent data structures with certain properties and so provide a channel for communicating information about data structures.

Viewing these cultural units as being defined by convention entails that they are not necessarily fixed in meaning. The use of expressions evolve over time, and so also will the connotations associated with a particular term or expression. As well as varying over time, there will also be variations in the way individuals use and interpret expressions. Even in the case where rigid definitions exist, for example dictionary definitions or formal definitions of programming constructs, the usage and hence the 'meaning' of particular expressions are still capable of undergoing evolutionary change.

The evolutionary nature of any culturally based system of signification has implications for the possibility of formalising the content of such systems, and hence for formalising any domain such as that of software design. This has implications for the work done to attempt automatically to redescribe source code since much of this work involves attempting to formalise some of the 'knowledge' used in software design.

4.4.1 Ambiguity and context

One of the frequently cited properties of natural languages is ambiguity. One particular reason for the use of formal languages is that they are intended to reduce this ambiguity by providing a rigid denotation for the expressions of the language.

One reason for the ambiguity of natural language stems from the polysemic nature of many of its expressions. Thus a single term, eg "ball", can be interpreted as referring to a content in more than one semantic field. This ambiguity of meaning is reduced when a term is used within a text or a sentence since the surrounding material provides a context for the interpretation of the term. The interpretation of such a term can be said to be *context dependent* in that its denotation depends upon its context of use.

In contrast, the denotations of terms from formal languages are fixed by the definition of the language. So for example, in a formal language a fixed denotation can be given to the expression "IF a THEN b", independent of its context of use. The meaning of this construct *within* the formal system has been *decontextualised.* However, this decontextualisation does not render the interpretation of a formal text as being similarly context free. The interpretation of a text is not only a function of the text's denotation, but also involves connotation.

As a text is read, the reader is constantly creating expectations and hypotheses about possible interpretations. These expectations may be described as occurring in the form of contextual frames. These frames provide the mechanism through which simple expressions can be interpreted as part of a larger structure. The granularity of this context can vary from the particular language being used, down to the immediate neighbours of a particular term. A large part of interpretation work involves the selection of an appropriate contextual frame with which to continue interpretation.

The selection of appropriate contextual frames is not just the mechanism through which interpretation occurs, they *are* the interpretation. The text structures of which individual expressions are a part, are identified within text as instances of the system of shared codes which enables communication. Any particular interpretation of a text involves establishing many interconnected correlations between expression and content. It is these correlations that create the many connotations associated with a text, and it is these connotations that allow the text to escape from the page on which it is written and stand for something other than itself.

The correlation of these text structures with a content is not usually established by formal means. Even when a formal definition for this content does exist, for example a definition for a sort routine, it is not this formal definition that is used in practice to provide the expression with a content. This content will depend upon the totality of the interpretation applied to the text. Thus while a formal language may be used to fix the *denotation* of a text it cannot similarly define the *connotation* of the text, although certainly the denotation will constrain the choice of adequate interpretations.

4.5 Source Code as Text

At the beginning of this chapter I stated two formalist assumptions upon which much of the work in redescribing source code has been based. Firstly that many researchers implicitly assume that the 'meaning' of a piece of source code can be captured in a formal program model, and secondly that the problem of producing an appropriate redescription of code can be reduced to a problem of ensuring a formal correspondence between the source code and the new description.

The previous sections have tried to demonstrate some of the difficulties surrounding these assumptions that are caused by viewing descriptions of any form as representational devices (in particular as texts) rather than as formal objects. These observations lead to the suggestion that source code, designs, specifications and other products of software development should be viewed primarily as *texts* rather than as formal objects. This is not to suggest that these artifacts should not be analysed for their formal properties, but that it is important to remain aware that these formal properties are subsidiary to the role of these artifacts as representational devices.

The following sections aim to show the way that these formalist assumptions have been incorporated into much of the research work directed toward redescribing source code, and some of the practical effects of the embodiment of these assumptions in actual systems. In practice much of this is part of a much larger argument to show the way that the term 'program' has tended to become identified with some mathematical description that is capable of acting as an oracle, providing absolute answers to any question about the behaviour of a computer system.

4.5.1 The meaning of source code is not a formal program model

A programming language can be given some formal semantics which model the intended action of the code when run. In this way the source code can be translated into a mathematical model of the code's anticipated operation. This model will be termed here a *formal program model.* This model is often considered to be able to capture all the properties of the original source code. This formal program model is not always explicitly constructed, however, the assumption that the 'meaning' of source code is contained within this model still forms a basis for the code analysis that is performed.

The reason for terming this a formal program model as opposed to simply a formal model is to distinguish two specific properties of this model. Firstly, formal applies not only to the nature of the model itself, but to the fact that it is constructed using only formal information in the source code. Secondly, the use of the term program indicates that this model is in some way intended to capture the essence of what the source code (or higher level description) is referring to.

The term program is frequently used to refer to some implicit kernel of the process of software development. However, the term program is used in many different ways by different people (see Fetzer 1988 for examples). Within many of the approaches to redescribing software, the term program is frequently used synonymously with that of the formal program model, thus suggesting that such a model somehow captures the essence of the software without actually explaining why this is.

Viewing the intended object of source code and other products of software engineering (such as designs and specifications) as a formal program model is typically associated with researchers involved with formal methods. This view is well summed up by one of the main proponents and originators of the formal methods approach to software engineering, C.A.R. Hoare, when he states that:

> "Programming is an exact science in that *all* the properties of a program and *all* the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself from purely deductive reasoning." (Hoare 1969, italics added)

The view of researchers such as Hoare and Dijkstra as illustrated in the above quote is that by producing a formal program model from the program text (ie the source code) then the ‘program’ itself becomes a mathematical object, even though this clearly ignores the importance of commenting and labelling when writing "computer programs".

A similar identification is apparent in approaches to redescribing source code, and particularly transformational approaches. For example

> "We define the semantics or 'meaning' or 'effect' of a program to be a function which maps the initial state to a set of possible final states" (Ward 1989, p3)

similarly

> "...once a formal representation of the software has been obtained properties and theorems about the program can be more easily derived..." (Lano and Breuer 1990, p22).

Plan based approaches are similar, both in the way all their analyses are based upon properties of a formal program model constructed from the source code, and also from the nature of the plans which they attempt to recognise in code.

> "Rather than dealing with a program in its source code form, the Recognizer uses the Plan Calculus representation for programs... In the Plan Calculus, a program is represented by an annotated directed graph, called a *plan*" (Wills 1990, p116)

This quote clearly illustrates the way that a "program" as a formal program model is considered to exist independently of any particular representation (ie source code or the Plan Calculus) and can be easily translated from one representation to another without any loss of content. This translation can be performed (indeed can only be performed) by concentrating on the formal information present in the source code and by excluding most of the non-formal information.

All other plan-based approaches also initially transform source code into some representation that is based only on formal properties of the original source code. Such a representation still contains potentially useful non-formal information in the names of the identifiers, but none of these approaches have attempted to use this information. Also in this initial transformation all comments and labels are discarded and so are not even potentially available to any analysis.

Only the approach of Biggerstaff (Biggerstaff 1989) seems to recognise the potentially rich nature of source code. His reuse-based approach to redescribing source code is the only one that does not intend to base code analysis on an impoverished representation of the original code based purely on its formal denotation.

> "we can see that connotation plays an important role in the process by which people deal with, interpret, and understand programs." (Biggerstaff 1989, p41)

The discussion earlier in this chapter argued that it is not possible to move from one representation to another, ie from source code to formal program model, in a neutral and objective way since the old representation becomes the 'object' that the new representation aims to describe. This process of description is always analogous to a process of map making, involving choices as to what features of the original to highlight, scale or repress in the new description.

Whilst it is clearly necessary and advantageous in many situations to translate source code into a new form which facilitates a particular type of analysis, none of the work in redescribing source code makes it clear that in performing this translation much information is lost from the original source code, information that cannot easily be recovered. This is despite considerable experimental work into human program comprehension which suggest that the understandability of code (and presumably of other derived descriptions) is a consequence of the interaction of a number of different features of the representation (eg Gilmore and Green 1988; Sheil 1981; Tenny

1988; and Oman and Cook 1990). This feature of source code is again something that would naturally be associated with source code by considering it as a text.

The difficulties associated with this view that one can translate a 'program' from one form to another without losing essential properties of the original source code are manifest in the reluctance of approaches based upon this assumption to use, or even consider as valuable, much of the non-formal information associated with source code. This is because such features of the source code are not considered to be part of the 'meaning' of the program as they do not form part of the formal program model.

For example, in a demonstration of how transformations can aid program analysis Ward explicitly includes a heuristic to remove any labels present in the code (Ward 1989, p16). This information is discarded since one of the goals of this approach is to make unstructured code (in Dijkstra's sense) structured. However, names of labels can be a rich source of information about the intention behind a piece of code, whilst there is little evidence to suggest that code that is restructured in this algorithmic way is in general easier to understand than the original in all but extreme cases of tangled code. If source code is seen as a text, then clearly the structure of the text itself is being altered in a way that has little regard for the original intention of the text. This will almost inevitably lead to a loss in the comprehensibility of the source code.

Ward states that later in the process mentioned above it is necessary to use information about the purpose and domain of the program to give a usable specification. This is precisely the sort of information contained in labels and immediately discarded by transformational approaches. Similar examples could be provided for many of the other approaches to redescribing source code and for many reverse engineering tools in general.

4.5.2 The importance of pragmatic considerations

The second formalist assumption associated with work on redescribing source code involves a concentration on the correspondence between the new description and the original source code to the exclusion of considering pragmatic features associated with these descriptions. This coupled with the assumption that the meaning of these descriptions is to be identified with some formal program model leads to more difficulties. In general this is symptomatic of a concentration on the semantic properties of a description rather than on the pragmatic features which render a description comprehensible.

Whilst the implicit assumption of a formal program model was exhibited similarly by both transformational and plan based approaches, this assumption is embodied in different ways by these approaches.

Dealing first with transformational approaches, this assumption is clear. For example the notion of a 'specification' as considered by these approaches is defined by Ward as:

> "We consider the "specification" of a program to be *any* equivalent program written in a very high level language (ie closer to mathematics)." (Ward 1989, p1, italics added).

and the mechanism for obtaining such a specification from source code, via an intermediate representations is formal and mechanical

> "The intermediate language...allows logical analysis and transformations into specifications" (Lano and Breuer 1990, p2).

This usage of the term "specification" seems at odds with the linking role of specifications described earlier in this chapter. In fact, the usage suggests that simply by expressing the function of the code in a high level, mathematical, language a specification results. This ignores the importance of such properties as comprehensibility and conciseness as components of a specification in favour of ensuring the correctness of the final specification defined as a correspondence with the original code.

What transformational approaches call "a specification" is actually an attempt to find a more explicit representation of the formal semantics of the code. This overlooks the fact that explicitness is not a mathematical property, but a property of a representation. The importance of the final form of the specification is implicitly recognised by these approaches, but these approaches consider that the formalisms themselves are capable of determining this final form, rather than the formalisms being a tool through which the representational goals can be achieved.

All transformational approaches are centered on a notion of program equivalence. This ensures that the final description is equivalent to the original formal program model and so ensures the correctness of the "specification".

If each transformation that is applied to the original code leaves an equivalent description then these systems are not making major alterations to the structure of the initial program model. If all transformed programs are equivalent in this way, then clearly the motivations behind such a transformation must be representational. However, little attention is given explicitly to addressing the communicative role of the final and intermediate products of the transformational process.

Transformational approaches are mainly intended to be operated in a semi-automatic fashion with the user guiding the choice of transformations applied to source code in a manner similar to that of program proof systems. In this way the user can supply the missing criteria of readability and understandability.

The library of transformations and parts of the system which automatically suggest transformations to apply can be seen as implicitly implementing a model of 'specification' readability. Unfortunately the implicitness of such a model will make alterations and modifications of this model very difficult to implement. This is important since for transformational systems to go beyond producing simple improvements to low level code, such as eliminating dummy variables or converting unstructured to structured code, these transformations will also have to include some domain specific knowledge.

This implicit domain model needs to be verified for a system to maintain its claim to produce 'correct' specifications from code. As Lehman points out (Lehman 1980) the correctness of a specification depends not only on the relationship between the code and the specification, but also between the specification and the domain that it is representing. For the specifications produced by a transformational system to be correct in the wider sense, this domain knowledge encoded in the transformations need to be capable of being checked against properties of the domain that it is modeling. The implicitness of the knowledge encoded in the transformations will make this difficult to accomplish.

Ultimately, the failure to provide an explicit domain model is caused by the failure to recognise the importance of pragmatic features of specifications. This will limit the ability of such systems to produce useful, genuine, code specifications.

Plan based approaches suffer different problems when faced with producing useful descriptions of code. Whilst plan based approaches all include explicit domain models and also recognise the importance of the final description of code being able to link the code with higher level domains, it is the psychological basis of these approaches which proves to be problematic.

Why should a psychological theory of programming be inadequate for approaches to redescribing software? The fundamental basis of plan based theories of programming as stated by Soloway and Ehrlich are that:

> "The basis of our approach is that expert programmers encode their higher level knowledge in the form of plans which represent many of the stereotypic actions in a program." (Soloway and Ehrlich 1984a)

Such an approach presupposes that there is relatively little variation between programmers in both the way they understand code, and in the nature of the plans in which their knowledge is encoded. Such an assumption is questionable. The view of source code as text given earlier in this chapter highlights the complex and interelated nature of source code when seen as a communicative device. Together with specific results of experiments (Gilmore and Green 1988; Davies 1990) this indicates that neither the process, nor the end results of program comprehension are generalisable across populations of programmers.

Rather than attempting to model the psychological processes which are involved in program comprehension, that is considering the *internal* processes that render a piece of source code intelligible, we should consider the way that a piece of code (or design) is used *externally* as a medium of communication by software engineers. The usefulness of a design level description of source code ultimately depends upon the ability of this description to communicate the operation of code across a community of software engineers, and not to any particular individual.

The psychological bias of plan-based theories of programming skill prevent these models from considering this pragmatic use of source code and designs. The characterisation of pragmatics given earlier in this chapter was that pragmatics is the study of the relationship of signs to the *users* of the signs. Since plan based theories concentrate on the individual and the internal rather than on the social and the external they are unable to fully grasp the role of source code and designs in software development.

A text based view of source code suggests that to understand the role of source code in communication we should attempt to identify the cultural units which are manifest in the code. Plan based approaches do not explicitly recognise this goal and in practice tend to get bogged down with excessive detail by attempting to postulate models which explain the states and processes which enable program understanding. Many of these states (often formalised as plans) have no clear correlate with any cultural unit that plays a role in communication since plan-based theories are primarily concerned with modeling internal rather than external processes.

It is only in reuse based approaches that the knowledge that is used for the analysis is based upon the way source code is described and characterised by a community of programmers. This is because these approaches explicitly perform a domain analysis of software design. This domain analysis identifies features related to cultural units and then attempts to formalise these features.

Unfortunately the nature of cultural units means that they do not easily succumb to formal definition. They only exist by way of a convention shared by a community and these conventions as subject to a constant evolution and so any formal definition of a cultural unit is necessarily an approximation. For example, given a concept such as <stack> we need to find some way to formalise a notion of "stackness" so that we can recognise an instance of this concept in source code. This vagueness of definition limits the degree of correctness that we can expect to achieve in redescribing source code.

4.6 Using Properties of Source Code as Text

The limitations of using only formal information in tools for redescribing source code and the importance of considering the role of source code and derived descriptions as a medium for communication between software engineers all stem directly from viewing source code as a text rather than as a formal object. This viewpoint suggests that in attempting to automatically redescribe source code we should pay more attention to features characteristic of source code's textual nature.

There has been considerable research effort directed towards automatic text analysis from the related areas of information retrieval, natural language processing and, to a lesser extent, knowledge acquisition. Although some of this research has informed current approaches to redescribing source code, this link has rarely been fully explicated and frequently has had relatively little practical influence.

This section aims to make some of these connections apparent, and to suggest some more direct ways in which techniques of text analysis could be used in the analysis of source code. It will consider briefly how techniques developed in the three fields mentioned above could be used to provide methods of analysing source code with the goal of producing high level information from the code, and to show where these techniques are related to some existing approaches to source code analysis.

4.6.1 Information retrieval

During the late sixties and early seventies, there was considerable interest in using the occurrence of simple lexical items or textual structures as a basis for the analysis of texts.

Most of this work was carried out within the field of information retrieval (van Rijsbergen 1979; Salton and McGill 1983). This field is primarily interested in developing systems to retrieve documents relevant to requests from users from large document collections. Most of the work in this field involved trying to find correlations between the occurrence of lexical items within a document with the perceived content of the document. Within this field a large number of different procedures for analysis were developed and evaluated.

The experimental work reported later in this thesis aims to apply techniques derived from research in information retrieval to demonstrate the potential use of non-formal information in source code. The approach followed will attempt to use the occurrence of particular terms within source code to suggest the possible function of the code. The use of techniques of text analysis developed in information retrieval research will be explored in more detail in the remainder of this thesis.

4.6.2 Natural language analysis

If we are considering source code as a text, then we should look at approaches to understanding natural language texts. If we look at work directed at the computerised understanding of narratives we see many approaches using the notion of plans as central to their analysis.

The development of scripts and plans as a means of understanding natural language text is in part responsible for the development of the theory of programming plans (cf 3.2.2). The use of scripts and plans to represent information about stereotypical situations for use in text understanding was initially developed by Schank and Abelson (Schank and Abelson 1977).

Within natural language texts, plans and scripts are used as methods of encoding high level structures to be identified in text. This is similar to the role they play in theories of program understanding, however, in natural language text analysis systems have been developed that implement the use of plans in a way distinct from their use in plan based approaches to redescribing source code.

FRUMP (DeJong 1982), was designed for summarising stories in newspaper texts, a task which can be seen analogous to the task of summarising source code to produce design level descriptions. This system used words or phrases found in the text to instantiate hypotheses about the content of the story in the form of a plan (actually a sketchy script). This plan created expectations about the more detailed content of the story and is used as the basis for further processing.

Further processing is achieved by attempting to fill the slots in the plan based on further analysis of the text, each slot having rules relating to how they may be filled. A refinement of the approach to text analysis taken by FRUMP was used to developing IPP (Riesbeck 1982). This differed from FRUMP in that it allowed predictions to be ranked according to their interest, and the processing used had a more bottom-up flavour than FRUMP.

This use of plans for text processing has two significant differences from the use made of programming plans in approaches to redescribing source code

- These systems use a *predictive* strategy, based upon the occurrence of particular words or phrases in the text to drive the text analysis. This has a number of advantages over the bottom-up strategies pursued by programming plan based system. Namely this strategy allows irrelevant or un-analysable areas of text (source code) to be ignored and it allows high level contextual concerns to be incorporated into the analysis at an early stage.

- The use of this predictive strategy also removes the need for complex grammars to be developed. One of the main reasons for the development of FRUMP and IPP was to attempt to process texts to a high level without the need for developing a detailed grammar to be used in parsing the text.

Both these features suggest that this style of approach may be of considerable use in analysing source code. In practice this predictive approach appears similar to that employed by Karakostas (Karakostas 1991) in his source code analysis tool, although no explicit reference is made to natural language approaches in his work.

4.6.3 Knowledge acquisition

Rather than attempting to redescribe source code using design and application level concepts, in keeping with the view of software reuse as a way of recovering some of the investment in existing systems we may wish to use the view of source code as text to provide a means of acquiring such domain knowledge directly from source code.

There has been some research on producing knowledge acquisition tools. These attempt to automatically acquire domain knowledge from natural language texts, mainly manuals and technical documentation (Frey, Reyle and Rohrer 1983; Nishida *et al.* 1986; Szpakowicz 1990). In general, these approaches use a small skeletal knowledge base containing high level information about the domain of interest to identify lower level domain structures within the code. This is done using specific lexical knowledge of the domain and using the structure provided by the text to uncover new domain knowledge. Much of this work is at an early stage, but the goals of the systems that Karakostas and Biggerstaff are developing seem very similar to this kind of work and may prove to have much in common.

Similarly, there is work in reverse engineering that aims to identify "design decisions" in source code using only formal information (Rugaber, Ornburn and LeBlanc 1990; Reynolds, Maletic and Porvin 1990; Selfridge 1990), and then to use this as a basis for the acquisition of new domain knowledge. These approaches might similarly benefit from considering some of the techniques used by approaches to knowledge acquisition from natural language texts, and also from considering source code as primarily a text itself.

4.7 Conclusions

This chapter has tried to present a view of source code as primarily a text. This view has been used as a basis for a critique of approaches to redescribing source code. Particular attention has been drawn to the failure of many of these approaches to recognise the importance of non-formal information in source code, and to appreciate the communicative role of source code and derived descriptions.

Non-formal information is not an embellishment of source code, designs, and specifications that enables the formal 'program' described by source code to be more easily grasped, but is an integral part of these texts. By way of providing empirical justification for this point of view, the remainder of this thesis describes the results of an experiment which aims to use non-formal information only to provide information about the nature of pieces of source code. This is not intended to suggest that the formal analysis of source code can be discarded, but to show that to consider formal information only is an unnecessary restriction on the range of source code analysis techniques.

In the final chapter of this thesis, the semiotic perspective introduced above is returned to to suggest some future lines of investigation for work in software maintenance.

Chapter 5
Automatic Indexing of Source Code

5.1 Introduction

The previous chapter made a case for using non-formal information in the analysis of source code. The rest of this thesis is concerned with investigating one particular way of extracting such information. This chapter introduces the basic ideas from information retrieval and related fields used in the experimental work that was undertaken. This work attempts to automatically index pieces of source code by utilising the non-formal information present in the code. The approach adopted is based upon viewing source code as a text rather than as a formal object, and using techniques of text analysis to obtain an indication of the function of pieces of source code.

Many approaches to software reuse take a similar textual view of source code and other reusable products of software development, considering these artifacts to be documents. These are documents that need to be described, stored and retrieved to enable reuse to occur. This is the idea of a library of software components (Frakes and Nejmeh 1986-87).

To effectively retrieve software components from such a library it is necessary to develop some form of classification scheme for software components, and to index components accordingly. This indexing of components allows the system to achieve effective retrieval.

For descriptions of components to act successfully as indexing devices, they must reflect the 'content' of the component. Thus a good description should present a high level view of the component, providing a link between the goals of indexing and redescribing source code.

Research in Information Retrieval has informed much of the work on software component libraries. One area of Information Retrieval research which has received only limited attention within the context of software reuse is automatic indexing. Automatic indexing aims to find ways of using the structure of documents and document collections to assign appropriate index terms to the elements of a collection, eliminating the need for the expensive manual indexing of documents.

Some of the techniques developed for automatic indexing have been applied in a limited way to software documentation (Wood 1987; Maarek 1989), however they have yet to be applied directly to source code. If these techniques were applied directly to source code they could generate high level information about the content of the code. It is the feasibility of this approach to source code analysis that is investigated here.

Specifically, the work reported here aims to use a classification scheme and an associated thesaurus as a basis for assigning descriptors to pieces of source code, the descriptors being indicative of the function of the code.

The motivations behind undertaking such an investigation are as follows:

1. Using non-formal information to automatically index source code according to function would demonstrate one possible mechanism for incorporating non-formal information into transformational and plan based approaches to redescribing source code.

2. Both Biggerstaff's (Biggerstaff 1989) and Karakostas' (Karakostas 1991) approaches to redescribing software rely in part upon utilising non-formal information in a manner similar to that being investigated here. The study presented here aims to provide some evidence as to the potential performance of such systems.

3. There is a need to be able to identify and index potential software components within existing source code to populate reuse libraries (see Basili 1988; Garnett and Mariani 1990). The approach suggested here should facilitate such a process.

5.2 Information Retrieval

A software component library is a form of document retrieval system. The performance of such systems has been widely investigated under the heading of *information retrieval* (see van Rijsbergen 1979; Salton and McGill 1983; or Heaps 1978 for an overview of this field). The interests of information retrieval can be characterised thus:

> "Information retrieval (IR) is concerned with the representation, storage, organisation and accessing of information items." (Salton and McGill 1983, p1)

A typical scenario for the use of information retrieval systems is as follows. Access to a large collection of documents relating to a particular field of interest (such as aeronautical engineering) is required by a group of users. An information retrieval system aims to allow documents to be retrieved from the collection which are relevant to particular, user formulated, requests. Information retrieval research is interested in finding ways to satisfy users' requests for information.

To be able to effectively retrieve documents from a collection, firstly a suitable *classification scheme* must be developed to arrange the elements of the document collection. These elements must then be *indexed* according to the classification scheme. The final performance of the system must then be *evaluated.*

5.3 Software Classification

Classification schemes aim to group similar objects from a given universe together into *classes*. There have been a number of approaches to classifying software for reuse, an overview of these approaches is given in (Albrechtsen and Boldyreff 1990).

Classification schemes can be divided into *enumerative* and *faceted* schemes. Enumerative schemes divide a given universe into successively narrower and narrower classes. These classes are usually arranged to display the hierarchical relationship between classes. Thus classification is achieved by breaking down a universe into smaller and smaller pieces. An example of such a classification is the Dewey decimal classification.

An alternative approach to classification is to use a faceted scheme. In a faceted scheme, a classification is synthesised from a small number of elemental classes or *facets*. A facet can be considered as a viewpoint or dimension on a particular domain.

Prieto-Diaz claims that faceted schemes are more suitable than enumerative schemes for classifying software components because

> "Faceted schemes are more flexible, more precise, and better suited for large, continuously expanding collections." (Prieto-Diaz and Freeman 1987, p8)

Prieto-Diaz has developed a faceted classification scheme for software components based on six facets.

1. Three functionality facets which describe what the component does:
   - Function - the primitive function of a component, eg add, move.
   - Object - the objects manipulated by the component, eg characters, lists.

- Medium - the medium on which the action is executed, eg keyboard, file.

2. Three environment facets which describe the environment in which the component performs its action

   - System type - the application-independent environment of the component, eg lexical analyser, file handler
   - Functional area - the application-dependent environment of the component, eg transaction processing, CAD.
   - Setting - where the component is exercised, eg advertising, car dealer.

The triple, *<function, object, medium>* can be used to describe the functionality of the code, whilst the other three facets can be used to limit the range of components retrieved with a given functionality.

Murray Wood (Wood 1987; Wood and Sommerville 1988) has developed a related approach to the classification of components. His classification scheme is based around the development of Component Descriptor Frames (CDF's) which describe the functionality of a component.

Rather than simply use the triple, *<action, object, medium>*, to describe the functionality of a component, Wood uses the *action* of the component to specify a *skeletal* CDF with which to describe the component. This skeletal CDF provides slots to be filled with appropriate descriptors. These descriptors broadly correspond to those that may be used in the *object* and *medium* facets of Prieto-Diaz's scheme.

However, the skeletal CDF form recognises that *actions* are not necessarily always specified by one *object* and one *medium* facet, but instead the action itself will usually determine the number and the role of the entities manipulated by the component. This makes clear the central role of *function* or *action* in describing a component. These CDF's are usually displayed diagrammatically, examples are given below in *figure 5.1* for the action *edit* and the action *communicate.*

$$\begin{tikzcd}
\text{EDIT} \arrow[r] & \text{object to be edited}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \text{object to be communicated} & \\
\text{source} & \text{COMMUNICATE} \arrow[l] \arrow[u] \arrow[r] & \text{destination}
\end{tikzcd}$$

Figure 5.1: Examples of Component Descriptor Frames

5.3.1 Formal specifications as component descriptors

As stated in *chapter 1* the original brief of this project was to develop tools and techniques for obtaining formal specifications from source code. Given the work that was subsequently undertaken it is worth taking a detour at this point to explain why formal specifications are not suitable for describing and indexing the components of a software library, and hence why no attempt was made to automatically index pieces of code with formal specifications.

The difficulty with using formal specifications as a basis for describing and retrieving components from a component library are caused by the *precision* of formal specifications. The precision with which a formal specification can represent a piece of code is the major advantage of formal specifications over other specification techniques, however this very precision is anathema to the goals of retrieval.

The aim of classification is to group like elements of a universe together. A suitable mechanism for describing a classification then needs to be able to capture notions of "aboutness" (Beghtol 1986). Formal specifications are unable to capture notions of aboutness since there are no general techniques available for either proving the semantic equivalence of syntactic variants of a specification, nor of defining criteria of "goodness of fit" between a specification and a component.

These problems, together with the effort involved in creating and maintaining such precise descriptions of components, render formal specifications unsuitable as a mechanism for describing and retrieving components from a software library. This is not to say that a formal specification is not of use in providing a precise description of a component once retrieved, but as a basis for retrieval formal specifications are highly unsuitable.

5.4 Vocabulary Control

Information retrieval systems can be divided into two according to the vocabulary they use for retrieval purposes (and hence indexing). Systems can either be described as *free text* systems or *controlled term* systems.

In free text systems there is no restriction on the natural language terms that can be used in queries or as document descriptors. In general, free text systems tend only to be applicable to areas where there already exists a well defined and precise terminology, for example areas of law or mathematics. In less well defined domains, such as software design, such systems suffer from problems of achieving *exhaustivity* in retrieval. That is, such systems fail to retrieve a large proportion of the documents that might be considered relevant to a particular request. This is because the term or terms used to specify the request will only occur in a small number of the relevant documents.

To increase the exhaustivity of retrieval in such cases, systems of controlled terms are used. The role of a controlled vocabulary is to "facilitate communication in the information retrieval process" (Lancaster 1979, p178). In such systems the number of terms that can be used in retrieval requests and document indexing is restricted to specific *index terms* only. The set formed by the union of these index terms defines the *vocabulary* or *index language* of the system. The vocabulary used by an information retrieval system has considerable effect upon the systems performance

In deciding upon which index terms to include in a vocabulary it is necessary to consider the intended role of such terms in retrieval. The aim of using a controlled vocabulary is to render the documents in a collection as dissimilar as possible, and so make it easier to distinguish between one document and another (Yu and Salton 1977). Thus a good index term can be defined as one that renders the documents of a collection more dissimilar, whilst a bad term decreases this dissimilarity. In this way the discrimination value of an index term can be assessed with respect to a particular document collection.

In empirical studies, Yu and Salton (Yu and Salton 1977) found a relationship between the discrimination value of a term and the frequency with which a term is assigned to the members of a collection (document frequency). They found that the best index terms (in terms of discrimination value) were those that were assigned to between 1-10% of the documents in the collection. Terms with a higher or lower document frequency than this tended to be poor discriminators.

However, most significantly for this study, this theory is used to suggest that terms with a document frequency of less than 1% can have their discriminatory power improved by including such terms in *term classes*. These term classes then have a higher document frequency than the individual terms contained within these classes. Hence using these classes as content identifiers rather than the individual terms leads to a higher discrimination value.

A common way in which terms are combined together into term classes is in a *thesaurus*. The use of a thesaurus is central to the work reported here since it is the thesaurus which ultimately provides the information necessary for the automatic indexing of source code.

5.4.1 The thesaurus

Lancaster describes a thesaurus in an information retrieval context as follows:

> "A thesaurus is a limited vocabulary of terms,...,(it) provides control over synonyms, it distinguishes homographs, and it brings related terms together." (Lancaster 1979, p181)

The thesaurus is a device used to provide vocabulary control. This is achieved by grouping related words together to form thesaurus classes, and by relating classes to each other. The simplest form of thesaurus is one in which terms are grouped together into discrete classes. This is often used to control problems in retrieval caused by considering synonyms or near synonyms as distinct terms.

Thesauri can be constructed in two ways, either manually or automatically. Automatically generated thesauri are produced by analysing the document collection for terms that increase the dissimilarity between classes of the collection. This approach is often associated with the automatic generation of a classification scheme for the document collection. The difficulty with such approaches is that although they can lead to reasonable retrieval performance, often the grouping of terms within the thesaurus appears to lack any real semantic cohesion. This is because the association of terms is based upon statistical properties of the document collection rather than semantic properties of the terms themselves.

The application of some techniques of automatic classification and thesaurus generation to collections of source code has been briefly investigated as part of the Practitioner Project (Boldyreff 1989). However, the results of this analysis revealed that the automatically derived thesaurus entries failed to correspond to the categories people use when describing such objects. As Boldyreff states in reference to an analysis of source code to identify suitable terms for indexing a collection

> "The terms used within the source code, whilst they might be usefully employed as indexing terms, do not appear to be descriptive of the source as a whole." (Boldyreff 1989, p2).

In contrast, manually developed thesauri are based on semantic relations between terms (based either on congruence of terms or hierarchical relationships). These thesauri are produced through performing an analysis of the domain of interest and so the groupings of terms used correspond to those used within the domain. This is consistent with the need for redescribing source code by using descriptions derived from understanding the way these descriptions are used as communication medium, as described in chapter 4.

For these reasons most approaches to software classification have relied upon the use of manually constructed thesauri to provide vocabulary control. It is only the use of manually developed thesauri for automatic indexing that will be investigated here.

5.5 Automatic Indexing

The problem being tackled here is to be formulated as an *automatic indexing* problem. Automatic indexing assigns *descriptors* to the documents of a collection. These descriptors are then used to stand in place of the document for retrieval purposes.

Automatic indexing can be performed in unison with the automatic development of a classification scheme and an associated index language (see Salton and McGill 1983, chapter 3; Sparck Jones 1971). However, the approach here is based upon the idea of there having been the prior development of a classification scheme and index language for reuse purposes and so these 'unsupervised' approaches to indexing and classification are not considered here.

The indexing problem can be viewed as a pattern recognition problem. We have an external conceptual system defined by some process (such as manual indexing) which is capable of assigning descriptors to documents. We wish to develop an automatic process to approximate the performance of this external system. To do this is it necessary to identify features contained within documents which correlate with the assignment of certain descriptors.

In this particular scenario the documents are pieces of source code, whilst descriptors correspond to high level descriptions of the code's function. Clearly this task has much in common with the approaches to redescribing source code described in the previous chapter. However, whilst the approaches described in the previous section relied mainly upon techniques of *syntactic* pattern recognition, such as parsing and graph matching, the approach taken here will be based on techniques of *statistical* pattern recognition.

Statistical approaches to pattern recognition firstly extract measurements from the object of interest and then use statistical properties of these measurements as a basis for assigning an object to a particular class (Schalkoff 1992). This often involves assuming some underlying state of nature that forms the basis for the generation of these patterns.

In practice, most approaches to automatic indexing are based on the occurrence of natural language terms or expressions in documents rather than the formal or structural properties of documents. In attempting to automatically index source code using non-formal information, it will similarly be terms and expressions occurring in the code that will be used for indexing rather than any formal properties.

5.5.1 The Darmstadt Indexing Approach

The probabilistic formulation of the indexing problem given here is based upon the Darmstadt Indexing Approach (DIA) (Knorz 1982; Fuhr 1989; Fuhr and Buckley 1990). This formulation considers the indexing task to consist of two steps, a *description* step and a *decision* step. In the description step, information about the relationship between a descriptor s and a document d is collected. This is information is captured in the *relevance description* $\mathbf{x}=x(s,d)$. This relevance description forms the basis for the decision step.

In the decision step the relevance description is used to estimate the probability $P(C \mid \mathbf{x})$ that given $\mathbf{x}$, assigning descriptor s to the document would be considered as correct, where correctness is defined by some external procedure (such as the results of manual indexing). This involves the development of an indexing function $a(\mathbf{x})$ to perform this estimation.

This indexing function can be considered as a specific instance of a more general discriminant function. To understand the use of discriminant functions in indexing it is necessary to describe some of the basics of decision theory.

5.5.2 Decision Theory

Decision theory aims to form a rational basis for making decisions based upon classifying and discriminating between objects. Given an object and a set of classes $\{\omega_1, \ldots, \omega_n\}$ to which the object may belong, a *decision rule*, δ, aims to assign the object to an appropriate class on the basis of some measurements taken on the object. These measurements are expressed as a feature vector $\mathbf{x}$.

A decision rule formalises the process of decision making by partitioning the description space, as defined by $\mathbf{x}$, into regions, $\{\Omega_1, \ldots, \Omega_n\}$ such that $\mathbf{x}$ is classified according to which region of the description space it lies in.

The description step that generates $\mathbf{x}$ is often referred to as feature extraction in pattern recognition, but these two processes can be considered as equivalent. (Whilst the DIA formulation above does not insist that the relevance description $x(s,d)$ should be a vector, we can assume that this will be the case without losing any of the expressive power associated with the DIA.)

The indexing problem can be formulated as a two class classification problem with classes ω_1 and ω_2 corresponding to the correct and incorrect assignment of a descriptor s to a document d respectively, based on the relevance description $\mathbf{x}$. Given a relevance description $\mathbf{x}$ for a given document d and descriptor s, we wish to find a decision rule δ that partitions the description space, Ω, into two regions, Ω_1 and Ω_2, such that if $\mathbf{x}$ lies within Ω_1 then s is assigned as a correct descriptor of d, whilst if $\mathbf{x}$ lies within Ω_2 then s is rejected as a correct descriptor of d.

The most fundamental decision rule is known as Bayes Minimum Error rule. This decision rule minimises the overall error associated with the decision regardless of the possible relative costs associated with making particular decisions. This rule assigns the relevance description $\mathbf{x}$ to the class that has the highest *posterior* probability, $P(\omega_i \mid \mathbf{x})$. That is, to the class that is most likely to be the correct class based on the value of the feature vector $\mathbf{x}$. Formally we can express this as

$$P(\omega_1 \mid \mathbf{x}) > P(\omega_2 \mid \mathbf{x}) \Rightarrow \mathbf{x} \in \Omega_1 \textit{ else } \mathbf{x} \in \Omega_2 \tag{5.1}$$

Estimating these *posterior* probabilities directly is often difficult but a more useful way of expressing this decision rule is obtained by applying Bayes theorem

$$P(\omega_k \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \omega_k) P(\omega_k)}{P(\mathbf{x})} \tag{5.2}$$

to give

$$p(\mathbf{x} \mid \omega_1) P(\omega_1) > p(\mathbf{x} \mid \omega_2) P(\omega_2) \Rightarrow \mathbf{x} \in \Omega_1 \textit{ else } \mathbf{x} \in \Omega_2 \tag{5.3}$$

$P(\omega_i)$ is the *prior probability* of an object belonging to class ω_i. This is the probability of an object coming from class ω_i before we make any observations. While $p(\mathbf{x} \mid \omega_i)$ are the class conditional probability density functions (pdfs). These are functions that represent the likelihood of observing $\mathbf{x}$ given that the object in question belongs to class ω_i.

In this 2-class case we can rewrite (5.3) in a more convenient likelihood ratio form:

$$\frac{p(\mathbf{x} \mid \omega_1)}{p(\mathbf{x} \mid \omega_2)} > \frac{P(\omega_2)}{P(\omega_1)} \Rightarrow \mathbf{x} \in \Omega_1 \textit{ else } \mathbf{x} \in \Omega_2 \tag{5.4}$$

The ratio on the right hand side of this inequality is the ratio of the prior probabilities. This expresses the likelihood of a piece of code being an instance of class ω_1 given that we have no information about the nature of the code. Whilst the left hand side of (5.4) is known as the *likelihood ratio*, since this expresses the likelihood of a feature vector, **x** originating from an instance of class ω_1.

Bayes Minimum Error rule implicitly assumes that there is an equal cost associated with deciding that **x** is a member of ω_1 or ω_2. However it is often the case that there is a greater *cost* associated with one misclassification (ie classifying an element of ω_1 as an element of class ω_2) than vice versa. For example, in indexing source code we are likely to be far more concerned about failing to index a document with a correct descriptor than we might be with incorrectly assigning a descriptor to a document.

The relative importance attached to each misclassification is formalised by a *cost function* c_{ij} which is the cost of misclassifying an element of ω_i as an element of ω_j. If we set the cost of making a correct decision to zero (ie $c_{11}=c_{22}=0$) then the revised version of the minimum error rule which takes into account the different costs associated with misclassification becomes

$$\frac{p(\mathbf{x}|\omega_1)}{p(\mathbf{x}|\omega_2)} > \frac{c_{12}}{c_{21}} \frac{P(\omega_2)}{P(\omega_1)} \Rightarrow \mathbf{x} \in \Omega_1 \ \textit{else} \ \mathbf{x} \in \Omega_2 \tag{5.5}$$

Far more conveniently, we can replace the right hand side of this equation with a single value, λ, which we can use as a *cutoff* value for our decision rule.

$$\frac{p(\mathbf{x} \ \ \omega_1)}{p(\mathbf{x} \ \ \omega_2)} > \lambda \Rightarrow \mathbf{x} \in \Omega_1 \ \textit{else} \ \mathbf{x} \in \Omega_2 \tag{5.6}$$

Since the value of all the prior probabilities for the descriptors will be considered to be equal due to problems of estimation (see next section) this considerably simplifies the problem. By varying the value of λ we vary the value at which we accept or reject a descriptor as a correct descriptor of a document. In practice this is equivalent to varying the relative costs associated with the two different misclassifications that can occur.

We can treat the ratio on the left hand side of the inequality as being equivalent to an estimate of $P(C \mid \mathbf{x})$, the performance of this estimate being evaluated by varying the value of the cutoff, λ.

The likelihood ratio, $p(\mathbf{x} \mid \omega_1)/p(\mathbf{x} \mid \omega_2)$, whilst not equal to the probability $p(C \mid \mathbf{x})$ that the decision step in indexing aims to estimate, clearly contains the same information. The greater the probability that $\mathbf{x}$ represents a correct indexing of a document, the greater the value of the likelihood ratio. In general any monotonic function of the likelihood ratio is equivalent in this sense. Therefore we can write our decision rule as

$$g(\mathbf{x}) > \lambda \Rightarrow \mathbf{x} \in \Omega_1 \; \textit{else} \; \mathbf{x} \in \Omega_2 \tag{5.7}$$

where $g(\mathbf{x})$ is a *discriminant function.*

The problem of finding an indexing function $a(\mathbf{x})$ is now the more general problem of finding a suitable discriminant function $g(\mathbf{x})$ to use in the decision rule (5.7). This discriminant function should, ideally, approximate the likelihood ratio (or a monotonic function of the likelihood ratio) since this gives optimum solutions to the decision problem.

5.5.3 Estimating parameters and non-informative priors

Most pattern recognition tasks are based around using a *design set* or *training set* of objects with known classification as a basis for developing a classifier, and then using an independent *test set* of objects for evaluation. The main use of the design set is in informing the development of the discriminant function $g(\mathbf{x})$.

There are two general approaches we can take to finding a function $g(\mathbf{x})$ with which to approximate the likelihood ratio. We can either try to estimate the likelihood ratio directly by estimating the probability distribution functions $p(\omega_i \mid \mathbf{x})$ directly from the design set, or we can use the design set to estimate the parameters of a more general discriminant function.

The training set is often used to estimate the value of various parameters used in the indexing process, for example estimating the value of the priors, $P(\omega_i)$, for the different descriptors or estimating the coefficients of a polynomial discriminant function.

This problem of estimating parameters highlights one of the main restrictions on the approach here, namely the size of the training set. The motivation behind the work undertaken here is to try to automatically obtain high level descriptions of existing code. For any approach to be of value, it must be possible to develop such an approach based upon a relatively small training set. If a large training set is necessary to enable effective performance then the development effort required would render such an approach useless.

This is a major constraint upon the development of an automatic indexing approach, particularly as applied to source code. For example, if we have twenty different descriptors of program function that we wish to apply to code these will tend to be mutually exclusive. That is, in general, a piece of code will perform only one main function that we will wish to index it by. The question to be addressed is how large a training set will we require to achieve a reliable estimate for the distribution of code functions across the general population (ie obtain a reliable estimate for the prior distribution of code functions)?

A rough estimate, based upon the optimistic assumption that examples of each code function are evenly distributed throughout the code and that examples of each function occur with equal frequency, suggests that a training set of more than 1000 pieces of source code would be necessary for reliable estimates to be obtained. (This estimate assumes that an estimate should be within +/-10% of the true figure, with a confidence of 90%). In practice the assumptions on which this estimate are based will not hold and an even larger sample would be required. Even so, 1000 pieces of code is still a large volume of code to examine in detail for the sole purpose of allowing subsequent code analysis. To make automatic indexing of code practical, a method of parameter estimation that uses a far smaller training set is required.

An alternative approach to estimation of priors to that applied above is to assume that we no *nothing* about the distribution of priors and assign values that favour no possible assignment of one descriptor over another. This means using a *non-informative prior* (see Berger 1980, pp82-90). In this case the use of a non-informative prior involves setting the value of all the prior probabilities, $P(\omega_i)$ equal. This is not the same as estimating that in the population as a whole instances of all the function classes are equally likely to occur across a particular population, it is simply using an estimate which introduces the least assumptions about the population as a whole into our calculations.

This use of a non-informative prior for indexing has empirical, as well as theoretical, justification in the work of Fuhr (Fuhr 1989). In this work it was found that setting all the values representing the probability of a document being relevant to an arbitrary request to 0.5 produced better indexing results on the test set than when values were assigned based upon the properties of the training set. Of course, this will not always be the case, but the lack of bias introduce by using a non-informative prior may often result in more reliable performance. The use of non-informative priors will be used again later to justify other parameter assignments.

5.6 Evaluation

The indexing problem is that of obtaining a document-descriptor relation for a set of documents, this relation representing those descriptors that are used to index particular documents. Evaluation attempts to quantify the ability of a particular indexing approach to rank each document-descriptor pair according to the suitability of the descriptor as an index term for the document.

In addition to being able to evaluate the effectiveness of an indexing strategy at providing a ranking of document-descriptor pairs, we also need to be able to interpret the results of this evaluation with respect to the differing goals of the research.

The motivations behind this work differ in their needs. To identify software components in existing code it is necessary to rank documents across a population of source code according to a particular descriptor, whilst for redescribing source code we wish to rank a single piece of code with respect to different descriptors. The approach to evaluation needs to be capable of being interpreted as related either to the retrieval of code relating to particular function across code, or to the ranking of document-descriptor pairs for a particular piece of code.

5.6.1 Recall, precision and fallout

The effectiveness of various strategies for ranking documents has been the focus of much research in information retrieval, mainly with respect to document retrieval. The most commonly used measures of retrieval effectiveness are the measures of *recall*, *precision* and *fallout*. The definition of these measures are based upon partitioning the document collection up according to the results of a request for documents.

A retrieval system can be thought of as determining a document-*request* relation. The document collection can partitioned with respect to a particular request into relevant and non-relevant documents. The collection can similarly be partitioned, according to the results of retrieval, into retrieved and non-retrieved documents. Recall, precision and fallout are then defined as follows:

$$Recall = \frac{No\ of\ documents\ retrieved\ and\ relevant}{No\ of\ documents\ relevant\ in\ collection}$$

$$Precision = \frac{No\ of\ documents\ retrieved\ and\ relevant}{No\ of\ documents\ retrieved}$$

$$Fallout = \frac{No\ of\ non\text{-}relevant\ documents\ retrieved}{No\ of\ non\text{-}relevant\ documents\ in\ collection}$$

The performance of a retrieval system is then often illustrated by calculating an average of these values over a number of different requests, as obtained at different values of the cutoff parameter which is used to control the number of documents retrieved. These results can then be displayed graphically either as a precision-recall graph or, less commonly, a recall-fallout graph.

The notion of relevance as used in retrieval is directly analogous to the role of correctness in indexing. In practice, both involve a subjective judgement as to the content of a particular document. We can extend the definitions of the above retrieval measures to act as measures of indexing effectiveness as follows.

The set of possible document-descriptor pairs can be partitioned into two sets representing correct, C, and incorrect $\bar{C}$, assignments of descriptors to documents in a way directly analogous to the partitioning of document-request pairs into relevant and non-relevant sets.

Similarly given a particular cutoff value, we can partition the same set of document-descriptor pairs into those that are accepted or rejected as correct descriptors for the document by a particular indexing rule. This partitioning is analogous to that of documents into retrieved and not retrieved. Using these partitions we can reformulate the measures of recall, precision and fallout as measures of indexing performance.

Given a set of documents (pieces of source code) D and a set of descriptors, S, with which to index the documents of the collection, we can define a set of all possible document-descriptor pairs $X = D \times S$. We can then partition this set, according to whether a particular element $x = (d,s) \in X$ represents an *a priori* correct indexing of d with s, into C and $\bar{C}$ as outlined above.

The indexing procedure that is being evaluated can also been seen as partitioning the set X into two subsets, I and $\bar{I}$ as follows:

$x \in I$ *iff s is assigned as an index term of document d*
$x \in \bar{I}$ *iff s is not assigned as an index term of document d*

This formulation leads to the following definitions of the three measures:

$$Recall = \frac{|C \cap I|}{|C|} \quad Precision = \frac{|C \cap I|}{|I|} \quad Fallout = \frac{|\bar{C} \cap I|}{|\bar{C}|}$$

Since there is a finite and small number of possible descriptors, we can consider the performance of an indexing procedure over all possible descriptors. This is unlike retrieval where we are considering retrieval performance with respect to a very large number of possible queries. Thus in the above definitions, the averaging of indexing performance over different descriptors is implicit in the definition. Indeed, the averaging strategy used implicitly here is equivalent to the strategy of *micro-evaluation* as used in retrieval experiments (see van Rijsbergen 1979, p150). This averaging strategy sees each document-descriptor pair as an independent test of the performance of the indexing function.

It can immediately be seen from the above definitions that the pair of measures most commonly used in retrieval evaluation, recall and precision, are not suitable for our purposes. These two measures are not 'parallel' in the sense that neither of these measures take into account the number of documents incorrectly indexed. This is a recognised deficiency of recall-precision output (Salton and McGill 1983, p176).

Recall and fallout are better in that recall measures the proportion of document-descriptor relationships that are correctly accepted by the indexing procedure, whilst fallout measures the proportion of document-descriptor pairs incorrectly accepted. It is recognised that as a measure of retrieval performance, recall-precision output is deficient in indicating the ability to reject non-relevant documents .

It was also found to be more convenient to introduce a new measure, based on fallout, which measures the proportion of document-descriptor pairs *correctly* rejected. This measure is called rejection and is defined as follows:

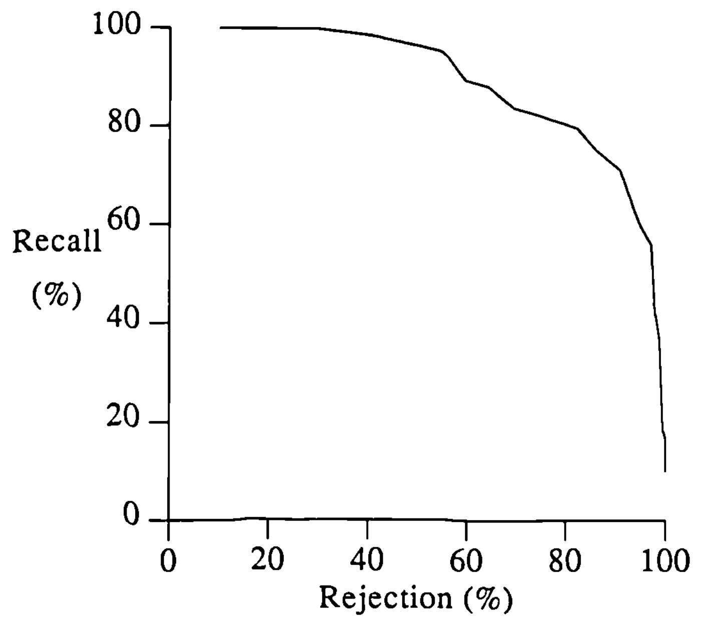
$$rejection = 1 - fallout = 1 - \frac{|\bar{C} \cap I|}{|\bar{C}|} = \frac{|\bar{C} \cap \bar{I}|}{|\bar{C}|}$$

This measure was introduced for the increased readability that it affords.

Using rejection instead of fallout allows the recall-rejection graph to be read from left to right as the value of the cutoff λ is increased as opposed to the less natural right to left reading necessary with a recall-fallout graph.

Intuitively, recall measures the proportion of correct assignments compared to the total number of possible assignments that are 'retained' by the indexing procedure, whilst rejection measures the proportion of incorrect assignments that are 'filtered out' by the procedure. Thus we can envisage the indexing procedure as a sieve which is intended to only allow the correct document-descriptor pairs through.

By varying the value of the cutoff, different values of recall and rejection will be produced. These can then be plotted to produce a graph. An example of a recall-rejection graph is given in *Graph 5.1*.



Graph 5.1: An example of a recall - rejection graph

The recall-rejection graph will form the main medium for presenting the results of the analysis. There is much information about the performance of a particular decision procedure missing from the graph, but generally, the closer the experimental curve is to the top right hand corner of the graph, then the better the performance.

5.6.2 Interpreting the recall-rejection graph

Because of the nature of this particular experiment, we can make a number of simplifications which make the results obtained easier to interpret whilst still allowing the results to be compared with those obtained from other information retrieval experiments. These simplification occur since in the majority of cases source code can be considered as performing a single function and hence is correctly indexed by a single descriptor. Further, this assignment of descriptors is relatively clear cut so we do not need to consider the problems involved in assessing degrees of 'correctness' or 'relevance' often associated with other information retrieval domains.

If we wish to consider retrieval performance, then we can realistically consider that most requests would be for code that performs a single function, rather than for the retrieval of a document that is relevant to a set of properties. Indeed, in a faceted classification scheme such as Prieto-Diaz's (Prieto-Diaz and Freeman 1987), even the extension of the classification scheme and indexing to other facets (such as *object* and *medium*) will not destroy this property since these facets are treated as disjoint.

In sum this means we can see the problem of indexing source code as far more of a pattern classification problem, where we wish to assign each piece of code to a single class, than is usually the case in information retrieval systems. However, it was not possible to initially frame the problem in this way since this formulation ignores the possibility that a piece of source code may be assigned none, one, or more correct descriptors, and it ignores the importance of ranking in the evaluation.

This approximation means that given a particular point on a recall-rejection graph, we can interpret that point as describing the misclassification rate of the indexing procedure. In this view we can use the recall measure to estimate the proportion of misclassifications due to failing to assign a correct descriptor to a particular document, whilst the value of the rejection measure can be used to estimate the proportion of misclassifications due to incorrectly assigning a descriptor to a document. Given additional information, these estimates can be used to estimate the overall misclassification rate for the indexing procedure and so provide a good indication of the overall performance of an indexing strategy.

5.6.3 Tests of significance

Given two recall-rejection graphs such as that it would be desirable to be able to demonstrate that the one curve shows significantly better performance than another. To do this would involve performing some statistical test of significance.

Unfortunately, because of the nature of the investigation, very few of the assumptions necessary for powerful statistical analysis of such data are met. Most tests (such as Students t-test and the Wilcoxon signed rank test) depend upon assumptions of normality and or symmetry. Neither of these assumptions can be considered as holding in the data under investigation here.

The only test that is considered suitable for this kind of data by van Rijsbergen (van Rijsbergen 1979, pp178-181) is the *Ordinary Sign Test* (see Gibbons 1985, pp100-106). Unfortunately this is a very weak test. Applying this test to two plots, a and b, on a recall-rejection graph involves comparing recall performance at fixed values of rejection along the graph. Values of + or

- are then assigned to each pair of values that show a significant difference. Pairs of values that are considered equivalent are discarded. The numbers of + or - values observed are then compared with a significance value based upon the possibility of such a distribution of + and - values occurring given that there is no significant difference between the two curves.

This test is only of limited use because of problems in selecting which points on the graph to use, and due to problems in deciding what counts as a significant difference between points on the two graphs. However, this does provide some indication of significance when graphs are close together.

5.7 Summary

Information retrieval aims to provide mechanisms for supplying users with information (usually in the form of documents) which is relevant to a particular query. Within software engineering the ability to retrieve documents derived from previous projects, such as source code and designs, and to be able to use these in new projects is seen as a major way of facilitating the reuse of these *software components*.

To be able to retrieve components effectively it is necessary to be able to classify them according to a software classification scheme. Current research suggests that faceted classification schemes are better suited to software than other approaches. Associated with such a classification scheme there is usually a *thesaurus*. This allows control over the vocabulary of the retrieval system by grouping related terms together to form *term classes*. This allows control over synonyms and in most cases will improve retrieval performance.

To allow retrieval, software components must be indexed according to the classification scheme. This involves assigning descriptors to the components. Ideally, these descriptors should be indicative of the nature of the component. The experimental work detailed in the following two chapters aims to use information contained in a thesaurus as the basis of an approach to the automatic indexing of source code.

The automatic indexing problem can be formalised using decision theory as a series of independent decisions as to whether to assign a particular descriptor to a piece of code. The success (or failure) of the indexing approach can then be evaluated in a manner analogous to that used in the evaluation of a retrieval system. The evaluation of the various indexing approaches investigated here primarily use the measures of *recall* and *rejection.*

Chapter 6
Experimental Design

6.1 Introduction

This chapter describes in more detail the experimental work carried out to automatically index pieces of source code. Much of this chapter is concerned with documenting the various indexing functions that were used to perform this indexing. This is necessary to allow the results obtained from the different approaches to be compared, and to allow these results to be compared with any future experiments into the automatic indexing of source code.

6.2 Aims and Assumptions

The investigation into the automatic indexing of source code using non-formal information was conducted using source code derived from IBM's CICS on-line transaction processing product. Code derived from this product was divided into two independent sets. One of these sets was used as a design set to develop discriminant functions that were used to index the code. The performance of these different functions was then evaluated using code from the test set.

Two of the indexing functions developed represented simple measures of thesaurus term occurrence within code, whilst the other indexing functions were used to compare two different approaches to the development of discriminant functions. One approach, the *probabilistic* approach was based upon modelling the underlying 'mechanism' that is responsible for the occurrence of thesaurus terms within code, whilst the *Generalised Linear Functions* (GLF) approach uses the data obtained from the design set directly to generate an indexing function.

The main assumptions on which the following work is based on are:

1. Statistical information about the occurrence of certain terms in a section of source code can be used to provide information about the nature of the code.
2. A piece of code can have zero (when no suitable class exists), one, or more than one correct descriptor. Code that has more than one correct descriptor corresponds to code that performs more than one function.
3. It is possible to develop a reliable indexing function based upon a relatively small design set.

6.3 The Source Code

The code used for the investigation was derived from IBM's CICS product. The CICS product has been on the market now for over 25 years and consists of a suite of programs to allow application programmers to build transaction processing systems. As such it is a complex and involved product which has been developed over a long period of time.

The system is implemented as a series of modules with the code contained in each module performing a number of related functions such as controlling access to data structures, controlling concurrency, and configuring different terminal types.

CICS is written in a mixture of 370/assembler and PL/AS. The source code used in this study was written in PL/AS. This a fairly low level procedural language similar to PL/1. PL/AS is what would be termed a 'structured' language apart from the following:

1. PL/AS allows multiple exit loops.

2. PL/AS allows 370 assembler to be interleaved in a PL/AS program through the use of a GENERATE statement. This allows for performance critical pieces of code to be written in assembler rather than the slower PL/AS.

It should be noted that whilst these features of PL/AS have little effect on the analysis performed here, the low level nature of much of PL/AS and embedded assembler code would make detailed formal analysis of the code very difficult to implement as part of a semi-automatic system for code analysis. The difficulties encountered in formally verifying such code, even when using verification tools such as SPADE, support this view (see Carre and Clutterbuck 1988).

6.4 The Classification and Thesaurus

A scheme for classifying pieces of source code from the CICS product according to the function performed was developed. This scheme was developed to be in accordance with Prieto-Diaz's faceted scheme (Prieto-Diaz 1985). In practice, only the function facet of this classification was implemented although this has subsequently been extended.

The classification provides 23 distinct classes which correspond to the function that a piece of source code might be considered as implementing. These classes are listed in *appendix 1*. The classification scheme was developed primarily through the examination of CICS source code and associated documentation. Other software classification schemes and consultation with CICS developers at IBM were also used in the development and validation of this classification .

Terms used to describe the function of a program were collected and grouped together to form a *thesaurus class*. These terms were derived from the design set, and also from general literature on programming and design and from the thesauri developed as part of the classification schemes of Prieto-Diaz and Wood.

Each class was assigned a *class descriptor*. It is these descriptors that are assigned to source code as result of the indexing process (manual or automatic). For example, the thesaurus class which corresponds to the programming function of performing a "search routine" is given below:

search = find, locate, look, scan, search, traverse

Each thesaurus class is intended to represent a particular cultural unit (see Chapter 4) which delineates the field of software design. These cultural units can frequently be considered as synonymous with the more colloquial 'concept'. Unlike some thesauri where a thesaurus entry is represented by a meaningless identifier such as a number, the classes here are intended to represent a cognitively meaningful unit.

One can consider each thesaurus class as having an *intention* and an *extension*. The intention relates the thesaurus class to the intuitive notion that the class is attempting to capture, for example the notion of <search> as common to all the pieces of source code that are considered as "implementing a search".

However, this intuitive definition is not sufficient to allow us to automatically assign source code to a class that describes the functionality of the code. We need a rule which allows us to determine whether or not to assign a particular piece of code to a particular class. This is what the approach to automatic indexing described here is attempting to do. It is trying to supply a rule, based upon the elements of a thesaurus class, that allow us to mechanically perform this assignation.

The decision was taken to use a classification for source code without any hierarchical relationships (ie no generalisation/specialisation relationships between classes), and to implement the thesaurus as sets of semantically related terms for the following reasons:

- This approach is in accordance with the more general faceted schemes for software classification developed by Prieto-Diaz (Prieto-Diaz 1985) and by Wood (Wood 1987). This compatibility is important since one of the assumptions on which the eventual applicability of this work is based is that the classification and thesaurus are likely to be developed as part of a domain model to enable software reuse. It is not expected that in general the classification scheme and thesaurus will be developed solely for the automatic indexing of source code.
- This classification makes evaluation of the results easier since either a piece of code is clearly either correctly indexed as an instance of a particular class or not. If a hierarchical classification were used, the problem of evaluation would be complicated because of the need to assess the accuracy of indexing in the case when code is indexed as belonging to a more general or more specific class than might be considered ideal.

6.4.1 The lexicon

Any term from the thesaurus can occur in many different forms in the program text, for example the term "add" may occur within text as "adds" or "added". To be able to recognise these alternative forms it is necessary to implement each term of the thesaurus as a set of lexical patterns that are searched for in the source code. There are two ways of ensuring that lexical patterns within source code are correctly identified as instances of terms from the thesaurus. We can either use a *stemming* algorithm to automatically normalise the terms that occur within the source code and the thesaurus, or we can manually implement a *lexicon* which explicitly associates terms from the thesaurus with patterns to be searched for.

Stemming algorithms can be used to reduce the variability associated with the terms occurring in texts. In general such algorithms remove suffixes to reduce words in the input to normalised stems, so for example "added" would be reduced to "add". These word stems can then be used as the basis for matching. This process is not error free since the removal of a suffix is not necessarily context free, so for example, one would wish to remove the suffix "ual" from "factual" but not from "equal". This problem can be partially overcome by introducing context specific rules to the algorithm although this adds to the complexity of the approach.

However, there is an additional drawback to using a stemming algorithm to identify thesaurus terms within source code. The frequent use of abbreviations within code and comments means that there are many non-standard ways in which a thesaurus term may occur. Many of these shortened forms would not be identified by a standard stemming algorithm. For example, the use of "defs" as an abbreviation for "definitions" or "tbl" for "table" would not be recognised by a stemming approach to matching.

The frequent use of abbreviations in code suggests that to recognise particular words it makes more sense to explicitly define a set of patterns that are searched for as instances of that particular word. This involves developing a *lexicon* which associates each word of the vocabulary of the system with a set of patterns to search for within source code. The relationship between class descriptors, the thesaurus, and the lexicon is illustrated in *figure 6.1*. The lexicon developed for this project is listed in *appendix 2* (this lexicon excludes words which are assumed by default to exist as a plural, by adding suffix "s", and in the past tense by adding the suffix "ed").

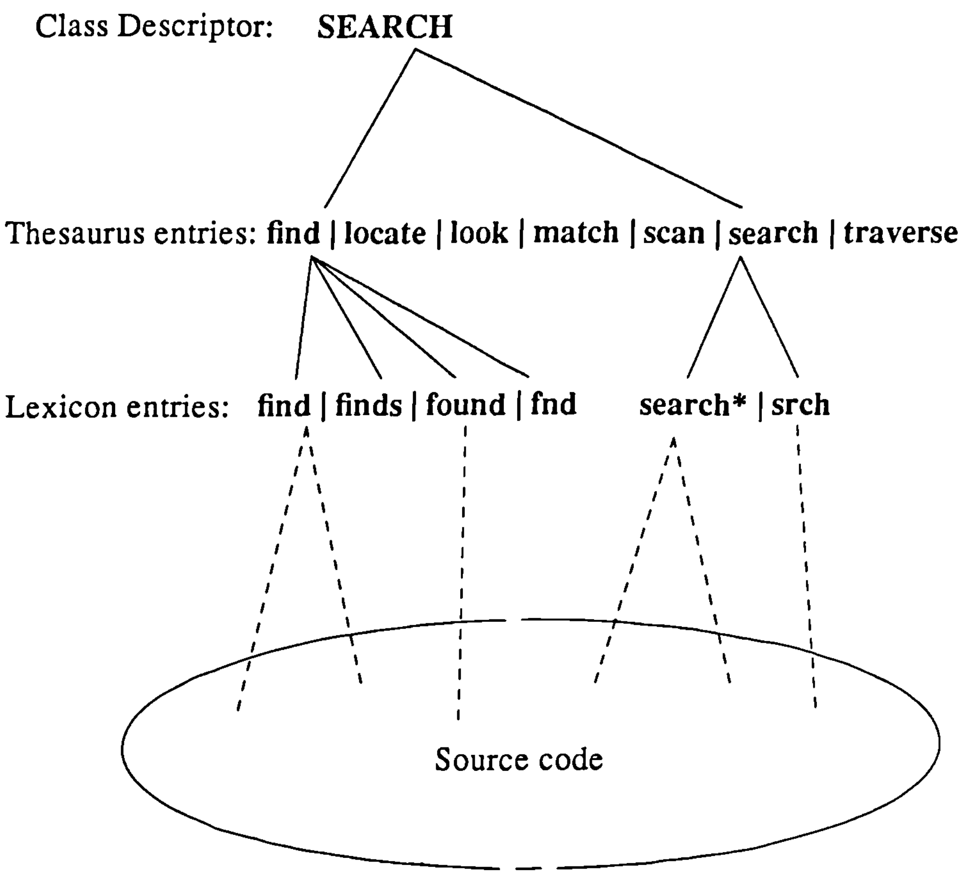

Class Descriptor: SEARCH
Thesaurus entries: find | locate | look | match | scan | search | traverse
Lexicon entries: find | finds | found | fnd search* | srch
Source code

Figure 6.1: The relationship between the thesaurus and the lexicon.

6.5 Representing Source Code

In section (5.5.1) the idea of a relevance description, x=$x(s,d)$, obtained from document s with respect to descriptor d was introduced as the basis for an indexing function. This section describes the form of the relevance description used in this project.

In Chapter 5 it was assumed that **x** was a vector. The form of the relevance descriptor used here uses a tree-like representation and so is not strictly speaking a vector. However for each indexing function described, **x** can be transformed into a vector and so for practical purposes there is no need to make a distinction between these representations.

The two basic pieces of information recorded within the relevance description, **x**, are to be

1. The number of *descriptor instances* found within the source code, *s*. That is the number of times that a thesaurus entry relating to the descriptor *d* is found within the code.
2. The length of the source code. This is defined as the number of *words* (alphabetic strings of more than two characters *excluding* program language keywords) contained within the code. It is only words that are potential descriptor instances and so the measure of code length used excludes any other expressions. This is so that in the probabilistic indexing functions, length correctly relates to probability of occurence of particular words.

Whilst for some of the indexing functions considered these two basic measurements are sufficient, for some of the measures we need to consider *where* in the source code descriptor instances have been identified. This is done by considering the basic block structure of the code.

6.5.1 Basic blocks and program trees

PL/AS, in common with most programming languages, has a well defined hierarchical structure. An example of some PL/AS code is given in *figure 6.2*. The structure of code written in PL/AS can be considered as consisting of a tree of basic blocks, with each basic block containing a linear sequence of instructions. This tree is known as a *program tree*. The notion of a basic block is frequently used in the program analysis techniques associated with compilers and program verification (see Muchnick and Jones 1981, p10).

In practice, the use of a basic block here is slightly different from the use made in other program analysis techniques since we explicitly wish to include comments and labels within each block, rather than discarding them. In practice this means that with each basic block derived from code we associate the comments that are on the same line as the program statements *and* any comments that precede the block. An example of part of the block structure derived from the example source code from *figure 6.2* is given in *figure 6.3.*

The approach taken here considers each basic block as an independent section of code on which a measurement is taken. Thus, rather than consider the source code as a monolithic whole, we can consider it as consisting of a tree of basic blocks, with an independent relevance descriptions being generated for each individual block. The set of these blocks then defines the relevance description **x** for the whole code. This use of basic block structure is not only of use in capturing structural information about the source code, but is also of use in implementing some of the calculations required for indexing functions.

Formalising this basic block representation, the relevance description **x** of a piece of code s consists of a set of nodes $A = \{a_0, \ldots, a_n\}$, where each node represents a basic block of the original source code and a_0 is the root of the program tree tree. This situation is illustrated in *figure 6.3.*

We can define three functions on A which encode the information about the source code and the program tree from which **x** is derived. These three functions g, s and t, all map an $a_i \in A$ to a positive integer. These enable us to represent certain attributes of each block of the tree as follows:

- $g(a_i)$, or more conveniently g_i, is the level of nesting of the node a_i in the program tree, with the level of nesting of the root node $g_0 = 0$.
- s_i is the length (as defined above) of the basic block represented by a_i.

```
/*This procedure searches a hash table for a key          */
HT_FIND: PROCEDURE;

    /*get the hash value   */
    CALL DT_HASH;
    INDEX = HASH_VALUE;
    SEARCH_LENGTH = 1;
    FIND_LOOP:
    /*Scan the hash table looking for an entry which  */
    /*matches the given key              */
    DO FOREVER;
        IF EMPTY(INDEX) THEN   /*empty slot*/
         DO;
         RESPONSE = EXCEPTION;
         REASON = NOT_FOUND;
         LEAVE FIND_LOOP;
         END;
        IF   DELETED(INDEX) THEN
         DO;
         IF DATA(INDEX) = KEY THEN /*key matches!*/
          DO;
          IF TMP(INDEX) THEN
             TRACE = ADDR(DATA);   /*save if nec*/
          RESPONSE = OK;
          LEAVE FIND_LOOP;
          END;
        END;
       IF INDEX < HASH_MASK THEN  /*wrap round*/
        INDEX = INDEX + 1;
       ELSE
        INDEX = 0;
       SEARCH_LENGTH = SEARCH_LENGTH + 1;/*inc search length*/
  END;  /*FIND_LOOP*/

END;  /*HT_FIND*/
```

Figure 6.2: An example of PL/AS code

- t_i is the number of descriptor instances associated with the basic block represented by a_i.

The complete relevance description $\mathbf{x}=x(s,d)$ then consists of a 4-tuple $\{A,v,s,g\}$

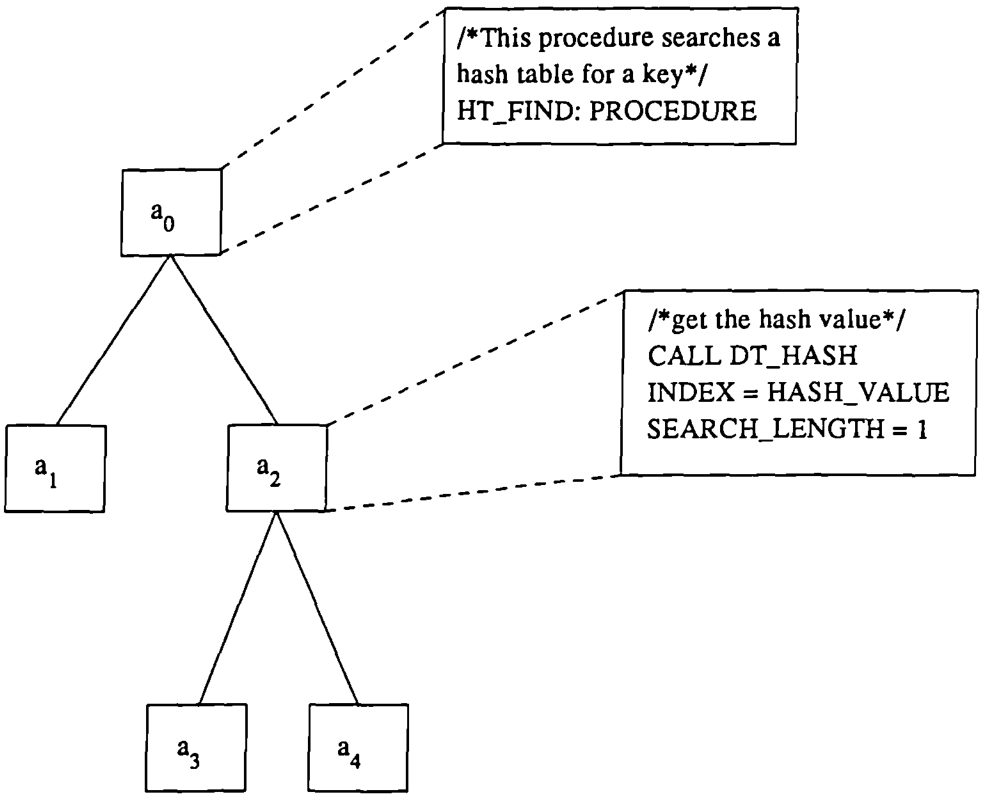


Figure 6.3: An example of block structure

From this representation we can easily recover the basic measurements of the number of descriptor instances and of code length by defining two new terms,

$$S = \sum_{i=0}^{i=n} s_i \tag{6.1}$$

and

$$T = \sum_{i=0}^{i=n} t_i \tag{6.2}$$

These terms, and the form of the relevance description described above will be used to formulate the different indexing functions used.

6.6 Implementation

The system used for experimentation was implemented using Intellicorp's KEE (Knowledge Engineering Environment) on a sun workstation. In practice, most of the code was written in LISP with KEE being used to provide support for object oriented programming and a graphical interface.

A simple parser was written which converted PL/AS source code into a tree of basic blocks as described in the previous section, each block being represented by a KEE unit (a unit is effectively a frame-like data structure).

These blocks could then be searched for the lexical patterns corresponding to descriptor instances, the number of such instances found then being added to the information contained in the unit. Code measurements could be extracted from this representation, or in some cases, indexing functions were applied directly to this tree structure. The values produced by applying an indexing function then formed the basis for evaluation.

The system was implemented to be an experimental system and as such the performance of the system was not considered as being a major issue during development. Since the main function of the system was to perform string searching and matching there is clearly scope for considerable optimisation. In practice the system performed at an adequate speed for the purposes of experimentation, taking about twenty minutes to process a module of about 7000 'words'.

6.7 The Indexing Functions

A number of indexing functions were developed to automatically index the design set. These functions were then evaluated on the test set to compare their effectiveness. The indexing functions that were applied can be divided into three groups.

- *The Simple Indexing Functions* - Measures of basic attributes of source code are used directly as discriminant functions.
- *Generalised Linear Functions* - The design set is used to estimate the value of the coefficients of a given function directly. The resulting function is then used as a discriminant function.
- *The 2-Binomial Model* - A model of the occurrence of descriptor instances within source code is described in terms of the Binomial distribution. This model of source code is used to develop an indexing function based upon the distribution of descriptor instances across the design set. This 2-Binomial indexing function is then extended in two ways; one that aims to provide a correction to this indexing function by taking account of terms that have a high document frequency, and one that aims to use information about *where* in the source code descriptor instances have been found.

6.8 The Simple Indexing Functions

Two simple indexing functions were used as a basis for comparison with the more sophisticated techniques used later, and in the case of the *keyword* function to provide evidence as to how the non-formal component of proposed systems for redescribing source code might be expected to perform.

6.8.1 Frequency

This indexing function uses the frequency with which descriptor instances occur within a piece of code as an estimate for the likelihood that such a descriptor is a correct index term for the code. Formally, this *frequency* indexing function is expressed as:

$$g(\mathrm{x}) = \frac{\textit{total descriptor instances}}{\textit{length of code}} = \frac{T}{S} \qquad \text{(F1)}$$

6.8.2 Keyword

Biggerstaff's and other approaches to redescribing software have suggested using the presence of particular keywords in code to generate hypotheses about the nature of the code. Specifically, they suggest using the comments that introduce a procedure or similar section of code to generate hypotheses concerning the function of the code.

To evaluate the performance of such an approach, an indexing function based upon the number of descriptor instances identified in the root node of the program-tree was implemented. This *keyword* function is defined as:

$$g(\mathrm{x}) = t_0 \qquad \text{(F2)}$$

The recall-rejection graph for this function consists only of a small number of discrete points, corresponding to cutoff values of $\lambda \geq 1,2,3,4,5$.

Although these points are joined together for clarity, there can be no significance attached to any points other than those actually derived from one of the above cutoff values.

6.9 Generalised Linear Functions

The use of generalised linear discriminant functions in pattern classification applications is well established (see Duda and Hart 1973, Chapter 5; Hand 1981, Chapter 4). This approach to pattern classification has also been applied to automatic indexing of standard document collections with good results (Fuhr and Buckley 1990; Fuhr 1989).

The technique applied here is one of a family of approaches which aim to use the design set to estimate the coefficients of a given form of discriminant function. These functions are defined to be linear functions of the elements of the feature vector **x**= $\{x_1,x_2,\ldots,x_d\}$, so the aim of the approach is to estimate the coefficients in a discriminant function of the form

$$g(\mathrm{x}) = a_1x_1+a_2x_2+,\ldots,+a_dx_d \qquad (6.3)$$

However, by transforming the feature vector into a vector of *functions* of **x** and by and by generating the corresponding measurements on the design set, **x** can be replaced by

$$\Phi(\mathrm{x})=\{\Phi_1(\mathrm{x}),\Phi_2(\mathrm{x}),\ldots,\Phi_1(\mathrm{x})\} \qquad (6.4)$$

In view of this transformation, we are no longer restricted to functions linear in the x_i. So for example we can consider polynomial functions of the x_i, or transformations of these elements, as our discriminant functions. Such functions are termed *generalised linear discriminant functions.* We must now describe how such functions can be estimated from the design set.

In Chapter 5 we stated that our decision rule based upon an indexing function $g(\mathbf{x})$ should be in the form of (5.7)

$$g(x) > \lambda \Rightarrow \mathbf{x} \in \Omega_1 \textit{ else } \mathbf{x} \in \Omega_2 \tag{6.5}$$

If we restrict the form of the indexing function $g(\mathbf{x})$ to being linear in the coefficients of the elements of $\mathbf{x}$, then we can use the design set to estimate the coefficients of this function. This is done by seeking to minimise the error associated with this function according to some criterion, the result of which is a discriminant function $g(\mathbf{x})$. This is the approach described briefly below.

Firstly, we need to rewrite the form of (5.7) so that we can express the problem of estimating the coefficients of $g(\mathbf{x})$ in the form of matrices. Firstly, we begin by attempting to find a linear indexing function

$$g(\mathbf{x}) = \mathbf{v}^T\mathbf{x}+v_0 \tag{6.6}$$

such that the decision rule (5.7) becomes

$$\mathbf{v}^T\mathbf{x}+v_0 > 0 \Rightarrow \mathbf{x} \in \Omega_1 \textit{ else } \mathbf{x} \in \Omega_2 \tag{6.7}$$

We can further simplify by defining

$$\mathbf{z}^T = (1,\mathbf{x}^T) \text{ and } \mathbf{w}^T = (v_0,\mathbf{v}^T) \tag{6.8}$$

We wish to find an estimate for the *weight* vector $\mathbf{w}$ using the elements of the design set. Given the decision rule above, a reasonable estimate would be one for which:

$$\mathbf{w}^T\mathbf{z}_i > 0,\ \mathbf{x}_i \textit{ is a member of } \omega_1$$
$$\mathbf{w}^T\mathbf{z}_i < 0,\ \mathbf{x}_i \textit{ is a member of } \omega_2$$

Where $\mathbf{z}_i^T = (1,\mathbf{x}_i^T)$ and the $\mathbf{x}_i$ are the elements of the design set.

To further simplify, we can define a new term $\mathbf{y}_i$ such that:

1. $\mathbf{y}_i = \mathbf{z}_i$ *for* $\mathbf{x}_i$ *a member of* ω_1

2. $\mathbf{y}_i = -\mathbf{z}_i$ *for* $\mathbf{x}_i$ *a member of* ω_2

Now our estimate of **w** requires that

$$\mathbf{w}^T\mathbf{y}_i > 0, \textit{for all } \mathbf{y}_i \textit{ derived from the design set} \qquad (6.9)$$

It is possible to estimate a value of the vector **w** using this inequality, however, the approach taken here aims to find a vector **w** which satisfies the set of linear equations

$$\mathbf{w}^T\mathbf{y}_i = b_i \qquad (6.10)$$

where the b_i are positive constants associated with each of element of the design set.

By setting $\mathbf{Y}=(\mathbf{y}_1,\mathbf{y}_2,\ldots,\mathbf{y}_n)^T$ and $\mathbf{B}=(b_1,b_2,\ldots,b_n)^T$ we can express the set of linear equations that we wish to solve in matrix form as

$$\mathbf{Yw} = \mathbf{B} \tag{6.11}$$

For a given vector **B** we can minimise the *least squares* criterion

$$(\mathbf{Yw}-\mathbf{B})^T(\mathbf{Yw}-\mathbf{B}) \tag{6.12}$$

to give a solution

$$\hat{\mathbf{w}} = (\mathbf{Y}^T\mathbf{Y})^{-1}\mathbf{Y}^T\mathbf{B} \tag{6.13}$$

Given a particular value of **B**, the value of $\hat{\mathbf{w}}$ can easily be computed using a multiple regression algorithm.

The approach used here used a value for the vector **B** which corresponds to solving the set of equations

$$\mathbf{w}^T\mathbf{y}_i = \frac{n}{n_1}, \textit{for all } \mathbf{x}_i \textit{ a member of } \omega_1 \tag{6.14}$$

$$\mathbf{w}^T\mathbf{y}_i = \frac{n}{n_2}, \textit{for all } \mathbf{x}_i \textit{ a member of } \omega_2$$

where n_1 is the number of design set vectors from class ω_1 (ie 'correct' relevance descriptions) and n_2 is the number of design set vectors from class ω_2 so $n=n_1+n_2$. The use of this vector is equivalent to calculating a solution according to Fisher's criterion (see Hand 1981, p82-84).

6.9.1 Generalised Linear Indexing Functions

The performance of two functions, with coefficients calculated from the design set as described above, was investigated. Both of these used the natural logarithm of the length of the code, rather than the raw length, since this improved performance.

The first function was linear in T and ln(S)

$$g(\mathbf{x})=a_1T+a_2ln(S) \tag{F3}$$

The second function was a quadratic in these two elements of **x**

$$g(\mathbf{x})=a_1T+a_2\ln(S)+a_3T\ln(S)+a_4T^2+a_5\ln(S)^2 \tag{F4}$$

6.10 The 2-Binomial Model

All the previous indexing functions have made no attempt to model the underlying process responsible for the occurrence of thesaurus terms within source code. The probabilistic model presented in this section aims to provide such a model.

Provided such a model is appropriate, an indexing function based upon the model should provide better and more flexible indexing. The disadvantages are that the development of such a model tends to require the use of a large amount of design data to obtain accurate estimates for the parameters in the model. In practice, to enable a reasonable model to be developed using the relatively small design set available here, a large number of simplifications are required to enable the value of these parameters to be estimated.

The first part of this section describes a model of thesaurus term production in source code. This is then used to develop an indexing function. Two further extensions of this indexing function are then proposed.

6.10.1 Modelling source code generation

The model described here considers the occurrence of thesaurus terms within source code to be the outcome of a *Bernoulli process*. A Bernoulli process consists of a series of *Bernoulli trials*. Each trial can be considered as an experiment which has only two possible outcomes, for example tossing a coin. The outcome of a series of Bernoulli trials is modelled by the Binomial distribution.

The model of the occurrence of index terms within source code presented here views the production of each word of the original source code as a Bernoulli trial. The production of each word is considered to be part of a 'random' process that generates the entire text. There is a probability p, that a word produced will be a member of a particular thesaurus class, and a probability (1-p) that it will not be a member of this class.

Given a particular value for p then the likelihood of finding x such terms in a piece of code of size n is given by the Binomial distribution:

$$p(x) = \frac{n!}{x!(n-x)!}p^x(1-p)^{n-x} \tag{6.15}$$

If we partition the set of relevance descriptions into the two sets C and $\bar{C}$ corresponding to correct and incorrect descriptors for a piece of code as in section (5.6) then we can suggest that the value of p responsible for the occurrence of descriptor instances will be higher for those $x \in C$ than for those $x \in \bar{C}$. This situation is shown in *figure 6.4*.

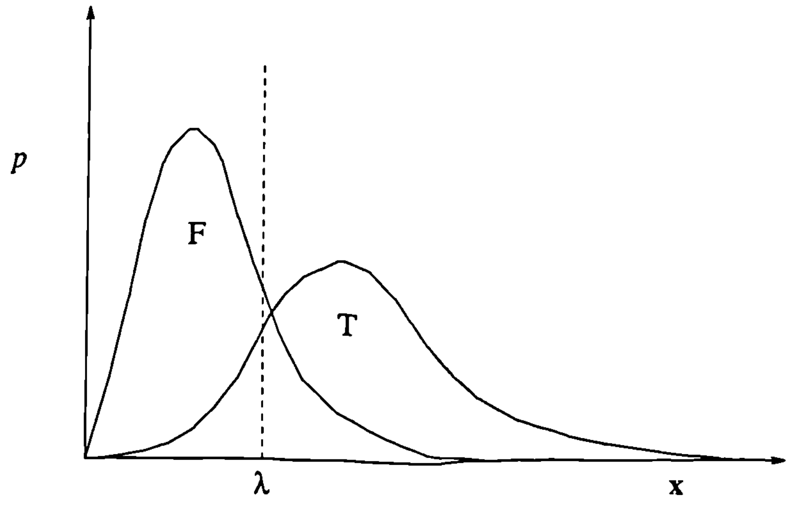


Figure 6.4: The two populations

If we can assume that the observed distributions of descriptor instances are consistent with such a process, and provide estimates of the two values p and q which represent the probability with which descriptor instances are produced within source code corresponding to the sets C and $\bar{C}$ respectively (this will be discussed later), then we can substitute the binomial distribution directly into the likelihood ratio (5.4) to provide us with an indexing function.

(Note: This modelling of the occurrence of terms within a document is very similar to that described in the 2-Poisson model that is often used in information retrieval (van Rijsbergen 1979, p29; Fuhr and Buckley 1990). The Poisson distribution is an approximation to the binomial for large n, but this condition is not always satisfied here since we wish to apply calculations to individual program blocks which may be of insufficiently large size.)

Substituting the binomial probability function in for $p(\mathrm{x} \mid \omega_i)$ in (5.4) and cancelling the binomial coefficient gives the following expression for the likelihood ratio

$$\frac{p^x(1-p)^{n-x}}{q^x(1-q)^{n-x}} \tag{6.16}$$

If we take the log of this we get a more convenient form to use as an indexing function $g(\mathrm{x})$

$$g(\mathrm{x}) = x\log\frac{p}{q} + (n-x)\log\frac{1-p}{1-q} \tag{6.17}$$

Rewriting this logarithmic version to calculate R at each node, and then summing the resulting values we obtain the following expression for the code as a whole

$$g(\mathrm{x}) = \sum_{i=o}^{i=n}\left[v_i\log\frac{p}{q} + (s_i-v_i)\log\frac{1-p}{1-q}\right] \tag{6.18}$$

It is easy to show that this expression is equivalent to evaluating (6.17) over the entire code since both are sums of sequences of Bernoulli trials.

6.10.2 Estimating p and q

The most detailed approach to indexing would involve estimating values of p and q for each term in the vocabulary of the indexing system. However this is not practical because of the restrictions we have placed on the size of the design set.

Rather than estimating the values of p and q for each word of the vocabulary, it is more sensible to conflate the terms and use values of p and q estimated for the class as a whole. It has been shown theoretically that the ability of specific terms to discriminate between documents is affected by the frequency with which such terms occur across the document collection (Yu and Salton 1977). For terms with a low document frequency (that is terms that occur in very few of the documents of the collection) Yu and Salton have shown that the ability of these terms to discriminate between documents can be improved by conflating the terms into a thesaurus class (*cf* 5.4). Thus, rather than attempting to estimate the values of p and q for each individual term, we wish to estimate the values of p and q for the thesaurus classes.

However, there is a difficulty with estimating these values with respect to a particular class based on the properties of the design set. The probabilistic model presented above assumes that the values of p and q remain constant regardless of the size of the code. This would entail that as the size of the source code increases, our certainty in any prediction about the nature would increase (ie the larger the piece of code, the 'easier' it is to classify correctly). Not only is this counter intuitive but measurements made on the design set contradict this assumption. In practice the measured value of p tended to decrease as the size of the code increased. To a lesser extent, q tended to increase as the size of the code increases but this increase was regarded as negligible. These results were backed up by the poor performance of a likelihood estimate based on (6.18) with fixed values of p and q.

Unfortunately, this makes estimating p and q for each thesaurus class infeasible since different estimates would have to be produced for different ranges of size of source code. This would lead to very small samples upon which to base any estimate and so a high degree of error would necessarily be associated with any such empirical approach. Instead it was decided to develop a model to explain the observed decrease in the value of p as size increases. This again necessitated considering all thesaurus classes as being identical in nature to enable the design set to provide enough information for parameter estimation.

6.10.3 Explaining the observed decrease in the value of p

Initially the decrease in the observed value of *p* as size increased was thought to be due to a reduction in the number of comments used as the size of code increases, with most comments occurring as header comments to describe the operation of the code and relatively few comments occurring within code. However, an investigation of the variation of the size of comments with the size of code conducted on the design set strongly suggested that there was no such reduction in comment "density" as the size of code increased (a linear relationship between code and comment size with a correlation coefficient of 0.997 was observed).

An alternative explanation for the reduction in the apparent value of *p* leads to a new model as to how descriptor instances are distributed within a piece of code. The model is based upon the assumption that the decrease in the value of *p* occurs because as the size of the code grows, a larger proportion of the code performs operations not directly related to the *main function* of the code, where the main function of the code is represented by the thesaurus class which would correctly index the code. (In practice the source code may implement more than one "main function" but including this possibility here would only cloud the discussion and make no difference to the model).

Since only the sections of code which directly implement this main function will produce descriptor instances related to this function with the higher probability *p*, then the frequency with which they occur over the whole code will tend to decrease as the size of the code increases.

If we consider source code as a program tree of basic blocks (see *figure 6.3*), we can view this tree as possessing a subtree which contains only those blocks of code that are directly related to the main function of the code. This situation is illustrated in *figure 6.5*.

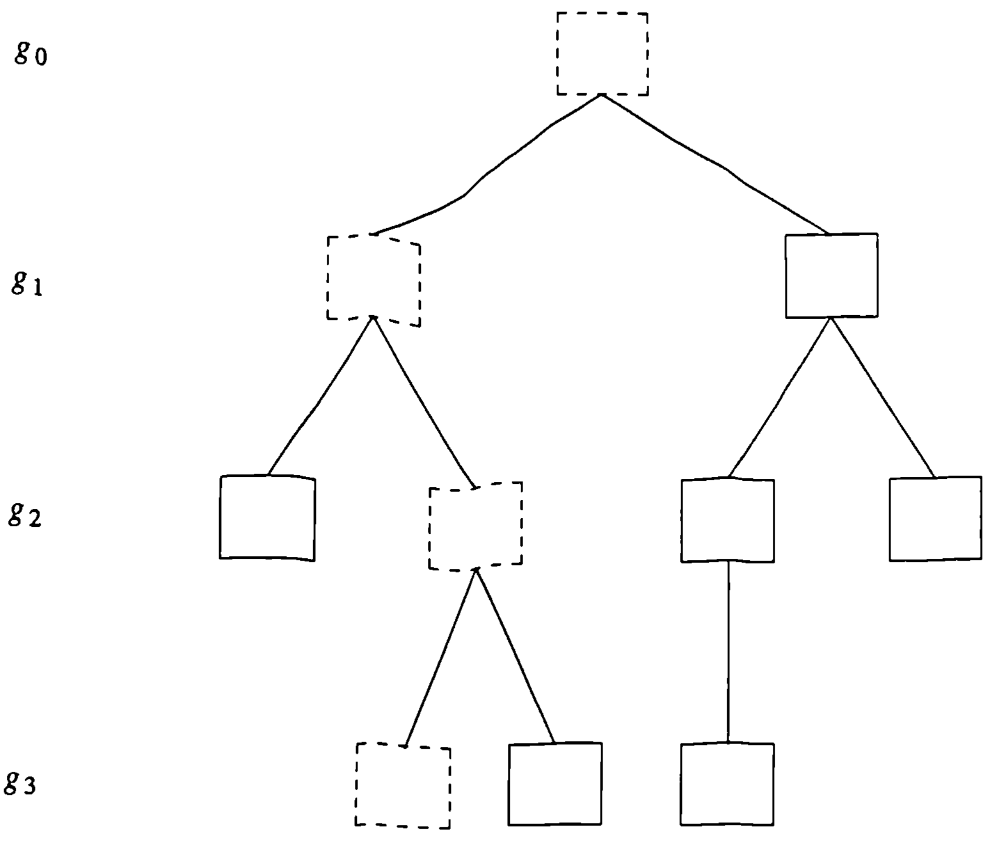


Figure 6.5: The "relevant" subtree

In this figure, the complete tree represents the code as a whole, whilst the tree which is marked by dashed nodes corresponds to those blocks of code that directly implement the main function of the code. As code increases in size, the ratio of the sizes of these two trees changes. This leads to a decrease in the observed frequency of descriptor instances relating to the main function of the code as the size of the tree increases.

We wish to find an expression for p either in terms of program size or the number of nodes in the program tree. Two assumptions simplify the resulting expression without making any significant difference to its final form. Firstly we can assume that the value of q is negligible (this is acceptable since q is considerably smaller than p and so its effect can be ignored). This allows us to disregard any contribution to the number of relevant index terms contributed by blocks that are not concerned with the main function of the code. Secondly we can assume that the growth of the sub-tree that implements the main function of the code (dashed boxes) is linear. This is not strictly speaking a correct assumption, but in practice it is the ratio between the growth rates of the two trees and not their absolute values that is significant.

Given these two assumptions we get the following estimate for p in terms of; the number of nodes in the program tree $n+1$, the mean rate of growth of the program tree r, and d the frequency with which descriptor instances occur within blocks that implement the function associated with that descriptor.

$$p = d\,\frac{\log((n+1)(r-1)+1)}{(n+1)\log r} \tag{6.19}$$

The value of d and r were estimated from the design set to be d=0.08, r=1.5. Using these vales to estimate p was found to produce reasonably good agreement with the observed frequency of descriptor instances from the design set.

6.10.4 The 2-binomial indexing function

The *2-binomial* indexing function is obtained by substituting (6.19) into (6.18) to give an expression for the likelihood ratio for a piece of code based upon the model of code generation described above. The value of q is assumed to remain constant as the size of the code increases. This gives the following expression for this indexing function

$$g(x) = \sum_{i=o}^{i=n} \left[v_i \log \frac{p}{q} + (s_i - v_i) \log \frac{1-p}{1-q} \right] \qquad \text{(F5)}$$

where

$$p = 0.08 \; \frac{\log(0.5(n+1)+1)}{(n+1)\log 1.5}$$

6.11 Term Weighting

Roughly speaking, the ability of a particular word to discriminate between instances of classes depends upon the relative frequency with which that term occurs in instances and non-instances of the class. The larger the ratio of these two frequencies, the greater the discriminatory power of the term.

Initially, the idea of weighting the contribution of terms to an indexing function to take into account the difference in the discriminatory power of terms was considered to be undesirable. This was due to the advantages to conflating low document frequency terms into a term class and because of problems in producing reliable estimates of these term frequencies. For these reasons, the values of p and q were left constant in the calculation. This decision can again be viewed as the use of a non-informative prior, and so an attempt to introduce the least bias into the indexing.

However it was noted that three terms from the thesaurus occurred far more frequently in source code whose function was unrelated to their respective thesaurus classes than other terms in the vocabulary. These words were "set", "initialise" and "validate". It was decided to weight these terms to reduce the influence that they had on the the 2-Binomial indexing function (F5).

The difficulty associated with weighting is that the estimates of p and q related to the frequency with which *any* term from the thesaurus class occurred, rather than to the frequency with which individual members of the thesaurus class occurred in code. Rather than readjust the system to deal with individual weight for each member of the vocabulary it was decided to implement the weighting by associating a weight c_i with words that were to be weighted, and considering the default weighting of all other terms to be unity. Then, for each thesaurus term that is present in the source code, there will be an associated weight c_i. If we use equation (6.17) then there will be precisely x such terms. The revised indexing function is as follows (with p calculated according to (6.19) again):

$$g(\mathrm{x}) = x\log\frac{p}{q} + (n-x)\log\frac{1-p}{1-q} + \sum_{i=1}^{i=x}\log c_i \qquad \text{(F6)}$$

In practice, this involves associating a value with each term we wish to weight, the contribution of this term to the final value of the calculation being reduced by this value every time that particular term occurs.

To estimate the value of c_i, note that this method of weighting is equivalent to substituting $c_i(p/q)$ for (p/q). Thus for the terms which we wish to weight, we can estimate the value of c by considering the increase in the value of q associated with the term. This leads to the values of $\log c_i$ shown in *table 6.1* being estimated from the design set.

6.12 Tree Weighting

The final indexing function evaluated uses the structure of the program code to weight the contribution of descriptor instances to the final ranking value assigned to a relevance description. This involves incorporating the assumption that terms closer to the root node in the program tree are likely to be more indicative of the code's function than those lower in the tree.

word	c_i	$\log c_i$
set	0.55	-0.6
initialise	0.6	-0.51
validate	0.65	-0.43

Table 6.1

The intention was to weight the whole tree, but primarily to *increase* the importance of the high level nodes in the calculation but without rapidly reducing the contribution of lower level nodes to a negligible level. This goal was implemented by weighting the value of the evidence contributed by a particular node in the program tree by

$$\frac{1}{g_i+1} \tag{6.20}$$

where g_i is the *generation* of node a_i, with the generation of the root node, $g_0=0$.

The choice of this function for weighting was based upon experimental results that suggested that this weighting was the more effective than an exponential denominator. This is prehaps because the proportion of the indexing value contributed by successive generations decreases slower than in the case of an exponential denominator although this hardly qualifies as an explanation. The choice of an effective method of weighting requires further investigation.

When applied to the probabilistic estimate (F5), this gives the following expression for the *tree weighted* estimate (with *p* defined as a function of the number of nodes in the program tree as in (6.19) again)

$$g(\mathbf{x}) = \sum_{i=o}^{i=n} \frac{1}{g_i+1} \left[v_i \log\frac{p}{q} + (s_i - v_i) \log\frac{1-p}{1-q} \right] \qquad \text{(F7)}$$

Chapter 7
Results and Analysis

7.1 Introduction

This chapter describes the results obtained from applying the indexing functions (F1-F7) defined in the previous chapter to code from IBM's CICS product. This is then followed by a more detailed analysis of the results of the experimentation.

7.2 The Design Set

The design set consisted of 155 pieces of source code of between 10 and 200 lines of PL/AS. These were derived from ten different modules that form part of IBM's CICS transaction processing system. The only criterion used for selection of code from the modules was that the code should be of suitable size.

Each piece of code was manually assigned to zero, one, or more classes according to the perceived function of the code. In most cases this assignment was relatively straightforward with the vast majority of elements of the design set being assigned to a single class which clearly corresponded to the function of the code. This manual indexing of the design set was then used to allow values to be estimated for the parameters used in some of the indexing functions.

7.3 The Test Set

The test set was constructed in a similar way to the design set and consisted of 149 pieces of source code. A comparison of some of the

	Design set	Test set
Number of pieces of code	155	149
Mean length of pieces of code	196	230
Pieces of code assigned at least one appropriate descriptor	148	122
Frequency of 'correct' descriptor instances in code	0.042	0.025
Frequency of 'incorrect' descriptor instances in code (all thesaurus classes)	0.020	0.013
Best misclassification rate (Tree-weighted F7)	0.12	0.26

Table 7.1

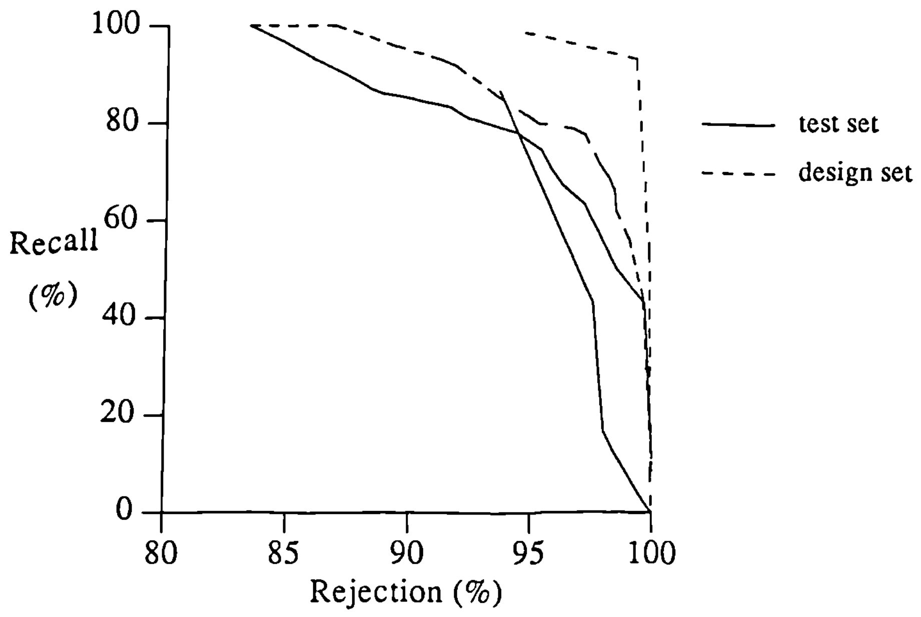
properties of the two sets are given in *table 7.1.*

There were many more members of the test set that could not be clearly identified (manually) as an instance of any of the classes of the classification, 27 as opposed to 7 for the design set. In these cases, the pieces of code were not considered to have any correct descriptors. However it was found that in both the design and test sets, whenever a piece of code could be assigned a correct descriptor, the code always contained at least one related descriptor instance. Thus all indexing functions apart from the keyword function (F2) could achieve 100% recall given a sufficiently low value of the cutoff λ.

Similarly, both the design and test sets had a minimum rejection rate of around 75% since the indexing functions were not applied for a particular thesaurus class unless at least one descriptor instance was found within the code.

A discussion of the results obtained on the test set are given below, before making some more general observations about the performance of the approach to automatic indexing of source code developed here.

7.3.1 The simple indexing functions



Graph 7.1

A comparison between the use of the *frequency* (F1) and *keyword* (F2) estimates on the design and test set are given in *graph 7.1*. In both cases, the keyword function is illustrated by the jagged line whilst the results of the frequency indexing function give a smoother curve. This graph illustrates the poorer performance of the indexing functions on the test set as opposed to the design set.

It also illustrates the way that the keyword indexing function (F2) is volatile in its performance. Although this function performed very well on the design set, performing far better than the frequency indexing function, its performance degraded considerably when applied to the test set.

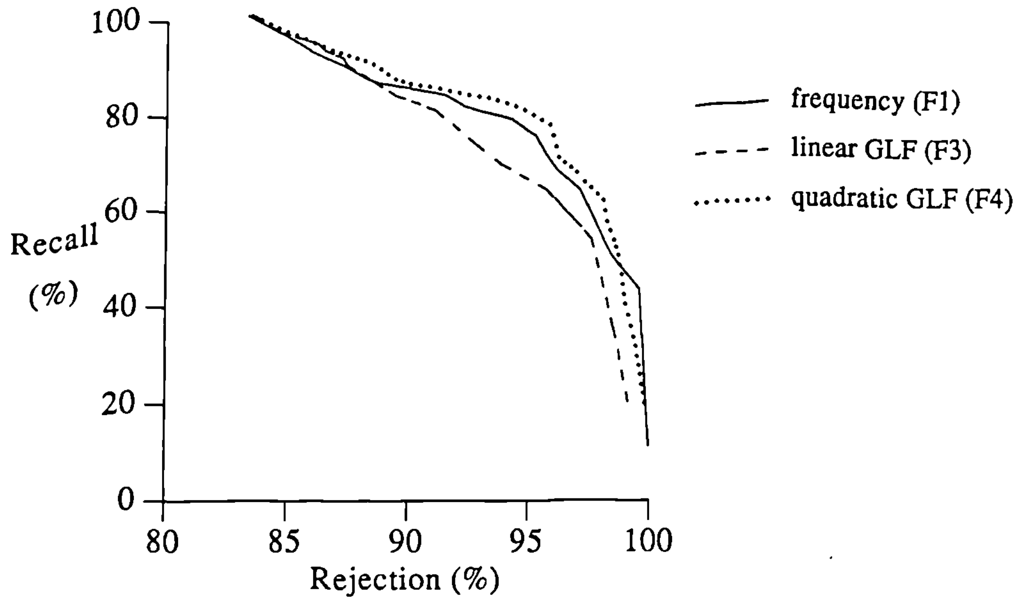
Perhaps most significantly, whilst the frequency indexing function could achieve a recall value of 100% on the test set the keyword function could only achieve a recall of 86.7% on the same set. This suggests that when analysing source code for the presence of terms that may be indicative of code function, analysing the entire code significantly improves the likelihood of locating a relevant term.

7.3.2 Generalised linear functions

Graph 7.2 illustrates the results obtained by applying the two indexing functions (F3) and (F4) to the test set. The results obtained by applying the frequency indexing function (F1) to the test set are also included for the purposes of comparison.

These results support those obtained by applying a similar approach to automatic indexing to more standard document collections (Fuhr and Buckley 1990; Fuhr 1989) in showing significantly better indexing performance obtained through the use of a quadratic rather than a linear indexing function.

The quadratic indexing function performed only slightly better than the frequency function. This was disappointing since on the design set, this function (F4) performed considerably better than the frequency function. This was because the increased average length of the code in the test set made apparent the poor performance of the indexing function on large pieces of code. Because of the low proportion of large pieces of code in the design set the discriminant function estimated from the design set did not perform well on code of large size.



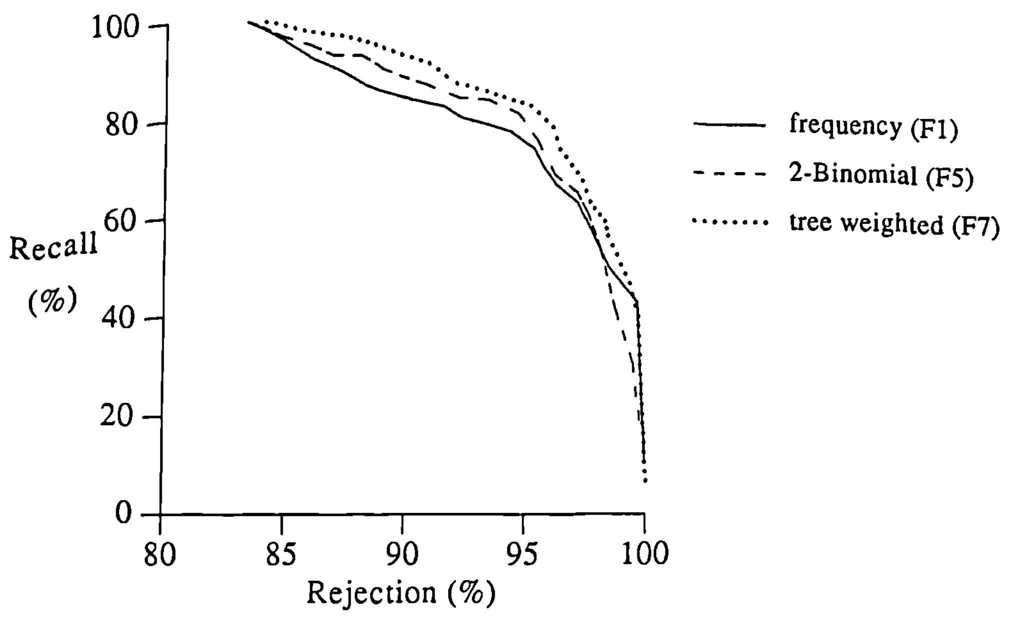
Graph 7.2

7.3.3 The 2-Binomial functions

Graph 7.3 displays the results for the 2-Binomial indexing function (F5) and the tree weighted version of this function (F7). Again, the frequency function (F1) is included for comparison.

Clearly, both these two functions perform better than the frequency measure, with the tree weighted version providing the best performance of all the indexing functions (significantly better than the unweighted version (F5) using the sign test at a confidence level of >99%). This result supports the hypothesis that terms found near to the root node of a program tree tend to be more indicative of the code's function than those found lower down in the tree.

In contrast, the term weighted refinement of the 2-Binomial approach (F6) was found to make very little difference to the performance of the indexing function. For this reason these results are not illustrated on a graph here. It



Graph 7.3

was not considered that an extension of this approach would lead to significantly improved performance, especially considering the effort required to obtain values for the term weights. (Indeed, one study of approaches to automatic indexing found that by replacing weighted values estimated from the design set by values equivalent to the use of a non-informative prior indexing performance was *improved* (Fuhr 1989))

In practice, the 2-Binomial approach proved to be relatively unaffected by differing values of the parameters used in the function. This is just as well since the frequency with which terms from the thesaurus were identified in source code was considerably lower for the test set than for the design set. By readjusting these parameters, better performance could be achieved on the test set although not to any major degree.

7.4 General Results

The graphs; *graph 7.1*, *graph 7.2* and *graph 7.3* illustrate the more significant results obtained from the test. In general the results obtained for the different indexing functions on the test set were comparable to those obtained from the design set, however, the performance of the indexing functions was significantly poorer on the test set. This is to be anticipated since the design set was used for the initial construction of the thesaurus and the indexing functions.

On closer analysis this poorer performance was related to two features of the test set:

- The average length of the code in the test set was greater than that of the design set (mean length of 230 words for the test set as compared with 196 for the design set). As the length of code to be classified increases, the difficulty of classifying it also increases. This feature of source code is predicted theoretically by the model proposed in section (6.9.3), and was also observed in practice.
- The frequency with which correct descriptor instances occurred in the test set was lower than that of the design set (independent of code size). There were two reasons for this; firstly that the vocabulary of the classifier was not extensive enough (that is there were some terms that should have been included in the thesaurus but were not), and secondly in the test set the conjoining of terms to make compound words (such as "addchar") was far more common than in the design set.

Although no formal analysis of the relative contribution of these two features was undertaken, the problem of compound words was far more significant in producing misclassifications. In practice it was found that the thesaurus was relatively adequate as is shown by the result that in both the design and the test set, *all* pieces of code were found to contain at least one descriptor instance from the thesaurus class by which the code was *a priori* indexed.

Unsurprisingly, the more sophisticated approaches provided better performance, with the tree weighted extension of the 2-Binomial model performing the best. In comparing the results of the GLF functions and the 2-Binomial functions it is necessary to consider the difference in development the two approaches take.

The GLF approach is very easy to implement and the development of such an indexing strategy is more straightforward than that associated with the 2-Binomial model. However, this ease of development also leads to a lack of flexibility of the approach with respect to the 2-Binomial approach. In a sense, this can be seen as a consequence of the large number of parameters available for 'tuning' in the 2-Binomial model, which makes the model very flexible although this does raise problems of obtaining values for these parameters.

Perhaps the most important distinction to be made is in the ability of the tree-weighted function to utilise information about *where* in a piece of code a descriptor instance has occurred. It is difficult to see how such information (which was included in the 2-binomial model as a built-in assumption about the nature of source code) could be utilised in a GLF style approach without leading to a large increase in the size of the design set required for this approach.

7.5 Analysis of Errors

The performance of an indexing function can be seen as a test of a decision rule which is used to assign a relevance description $\underset{\cdot}{\mathbf{x}}$ to one of two classes, ω_1 and ω_2 as described in section (5.5).

If $g(\mathbf{x})$ is the calculated value of an indexing function, then the decision rule can be seen as a test between two hypotheses. The null hypothesis H_0 that $\mathbf{x}$ was not derived from code that is correctly indexed by descriptor *d*, or the alternative hypothesis H_1 that $\mathbf{x}$ was derived from code that is correctly indexed by descriptor *d*.

We can then divide the errors of the decision procedure into errors of type I and errors of type II. Type I errors are those that result in the null hypothesis H_0 being wrongly rejected, that is a piece of code is erroneously assigned a descriptor *d*. Type II errors are those that result in H_0 being wrongly accepted, that is a descriptor *d* is wrongly rejected as being a correct index term for the source code.

The values of these errors at any particular point can be derived from the rejection-recall graphs since the type I error rate (%) is equal to 100-rejection whilst the type II error rate (%) is equal to 100-recall. The following provides some specific reasons for misclassification.

7.5.1 Type I errors

There were observed to be three main causes of type I errors:

1. The main reason for pieces of code being wrongly assigned a descriptor was due to a words from the thesaurus being located in code when the word is being used in a different context from that intended by its inclusion in the thesaurus. This often took the form of a verb intended to indicate the occurrence of and action being used as a noun to describe some data-structure related to the verb function.

   An example of this would be the use of the term "deletes" to describe objects that have been marked as deleted as opposed to signifying the operation of deleting some entity from a composite structure.

2. In some pieces of source code, sections of the code performed functions peripheral to the main function(s) of the code. In some cases this led to the code being incorrectly considered as implementing this peripheral function. This was particularly common in large pieces of code where much of the code deals with ensuring that preconditions are met before performing the main operation.

3. The final reason for this form of error came about through the common use of terms from the vocabulary of the system to describe aspects of the code's operation, or in some cases the mechanisms through which the main function of the code was implemented. An example of this sort of usage would be the following in the comments for a procedure, "/*This procedure sets the delete flag to '0'*/.

This final sort of error was generally restricted to the use of commonly occurring terms. It was the elimination of this *sort of error that lead to* an attempt to weight these terms accordingly in the *term weighted* function (F6), although this attempt was unsuccessful in practice.

The difficulty of using such frequently occurring terms (as shown by the results of this weighting) is that they tend to be very poor discriminators of the function of the code. However, removing such terms from the vocabulary of the indexing system leads to a reduction in the recall performance of the particular indexing function over the thesaurus class of which that term was previously a member. However, it will improve the rejection performance of the classifier over the other classes of the classification.

Yu and Salton (Yu and Salton 1977) suggest that such commonly occurring terms should be combined with other terms to improve their performance as document discriminators, however it is hard to see what terms one could combine to improve performance of indexing on source code.

Whilst in natural language text, the use of synonyms and near synonyms is commonplace to prevent the overuse of a particular word, within source code the terms used to describe particular operations are normally used consistently throughout a piece of code.

7.5.2 Type II errors

Errors of failing to correctly assign a descriptor to a piece of code again fell into three categories.

1. Descriptor instances were present and identified in the code but not in a high enough frequency to enable indexing.

2. Descriptor instances were present in the code but frequently were not identified as one of patterns specified by the lexicon. The most usual reason for this was that words were run together to form compound identifiers. For example, due to the workings of the string matcher, the token "add" would be recognised in "add_pct" it would not be recognised in "addpct", thus information that could be used in indexing is lost in this second version.

   In one module, used as part of the test set, nearly all the names of the procedures were in this compound and un-analysable form. This inevitably lead to poor performance (at least, poorer than possible) on this sample of code. It is difficult to see how this problem could be overcome without the implementation of a complex and time consuming pattern matching algorithm. To be able to split compound expressions such as "addpct" would require considerable information concerning specific naming conventions, and any implementation is likely to lead to side effects in the sense of introducing more incorrect matches into the indexing process.

3. Descriptor instances were present in the code but these words had not been included as part of the vocabulary of the system. In practice, this was rare.

7.6 The Effect of the Vocabulary on Performance

The effect of the vocabulary on performance was not studied directly, however the results of work on document retrieval systems are comprehensive enough to allow generalisation to this study. In considering how the vocabulary of the system could be extended, we must consider two different ways in which this extension may occur. Either new terms may be added to existing thesaurus classes, or entirely new classes may be added to the classification scheme.

The addition of new terms to an existing thesaurus class will tend to increase recall at the expense of the rejection rate. Clearly any increase in the vocabulary *must* improve (or at least, not diminish) the recall performance of the system. However such an increase in the vocabulary often leads to an increase in the number of type II errors, that is an increase in the number of pieces of code wrongly indexed by a particular class descriptor.

In this particular case however, most of the terms that are likely to be added to the existing classification scheme will be terms that occur with very low frequency across samples of source code, the more frequently occurring ones will already being included as part of the vocabulary of the system.

These new terms will be highly specific, only occurring in code where the code's function corresponds to that of the thesaurus class to which the term is a member. Thus recall is likely to be improved due to a small number of cases that are now correctly indexed by the enlarged vocabulary but were not so with the original thesaurus classes. However, the low document frequency of these new terms implies a high specificity, that is, these new terms are unlikely to introduce any extra "noise" into the system since they will probably only occur in the context of code that is correctly indexed by the relevant descriptor.

In the second case where new classes are added to the classification scheme, a different effect should occur. There is likely to be little effect on the recall performance (unless the new thesaurus class proves particularly good or bad at classifying code according to function), whilst there will be a considerable decrease in the rate of rejection associated with the automatic indexing.

The introduction of a new class is will probably introduce new terms into the vocabulary with a high document frequency. If this does occur then terms from this new class will significantly increase the amount of "noise" terms identified in source code. Individual pieces of code will contain descriptor instances from the new class, as well as descriptor instances of the pre-existing classes. This is will necessarily increase the number of type-II errors associated with any particular value of recall.

There are further problems associated with the expansion of the vocabulary through the addition of new thesaurus classes. These involve the way that as more classes are used to cover the domain, the classes are bound to become less disjoint and leading to problems caused by the occurrence of homographs, that is words with the same spelling but different meanings.

As the number of thesaurus classes increases, we can expect the conceptual distance (see Prieto-Diaz 1985, p115) between the classes in the classification to decrease. In practice this means we can expect to see the same or closely related terms present in distinct thesaurus classes. This will lead to an increase in the number of misclassifications of code through type II errors. Similarly, as more classes are added to the classification we can expect problems with homographs occurring in the vocabulary. An example of such a homograph might be the use of the term "add" to describe both a numerical calculation and an operation on a data structure.

These difficulties begin to define an upper limit to the accuracy that can be obtained by an approach to the automatically indexing source which is based upon statistical pattern recognition. Some of these problems could possibly be overcome by increasing the sophistication of the thesaurus as is outlined in the next chapter. In general though, the statistical approach is limited since it considers only the *raw data*. No attempt is made to consider the context of occurrence of certain terms (apart from their position in the program tree). Many of the problems associated with this kind of statistical approach are directly related to this failure to use context.

7.7 Comparison with Other Experiments

To further assess the performance of the approach to the automatic indexing of source code investigated here it would be useful to compare its performance with that of other, similar, experiments. Unfortunately, the only comparable experiments are a small and relatively informal study carried out by Wood as part of his work in developing a software components library (Wood 1987, p61) and a similar study by Maarek and Smadja (Maarek and Smadja 1989).

In his study, Wood used the presence of keywords in the description of Unix utility components to enter that component in the related class of the classification. Stemming software was used to improve the performance of his system in recognising keywords. He reports that about 70% of these components were inserted into appropriate classes although he does not report the percentage incorrectly entered into categories.

This compares with 86.7% of pieces of code from the test set entered in an appropriate class using the *keyword* estimate and 100% when considering the whole code. It is difficult to draw any conclusions here because of the informality and lack of information about Wood's experiment. However, this result does suggest that the use of thesaurus classes and the use of explicit lexical patterns to recognise index terms within text, as opposed to using stemming software, does improve performance.

The study by Maarek and Smadja (Maarek and Smadja 1989) was similar to Wood's. They attempt to index Unix components through the analysis of the associated manual documentation. Unlike Wood's approach which relied on the prescence of keywords, they make considerable use of lexical relations identified within the documentation as a basis for indexing. They report that their indexing approach produced significantly better retrieval performance than that achieved by the Unix "man -k" command but they do not provide any more detailed results than this.

7.8 Applications

The reason for conducting this investigation was to show that non-formal information can be made use of in tools for redescribing source code and in software reuse. The next two sections describe how the results of this work can inform these goals.

7.8.1 Application to redescribing source code

Perhaps the most significant result obtained from this study for approaches to redescribing source code is the observation that for both the design and test sets, in every case where the classification provided an appropriate descriptor for the code then a relevant descriptor instance was identified somewhere within the code. This result runs contrary to the common belief that non-formal information, and particularly comments, are unreliable.

It is accepted that this result is to some degree specific to this set of experiments since the code used was generally well commented, and that different code could produce different results. However this does show that in some cases non-formal analysis can be used to produce useful information from code.

The most likely use for the kind of information produced here is to generate high level hypotheses about the code's function. In this scenario, given a piece of source code to analyse a system for redescribing source code would first use non-formal information in source code to produce hypotheses about the code's function, possibly through the use of an indexing function similar to those developed here. These hypotheses would then be investigated in more detail by the system.

The results from the test set strongly suggest that an analysis of the entire code can produce far more information than considering only the root node of the program-tree. This latter approach is the one being suggested by Biggerstaff as a way of generating high level hypotheses about the code's function. By only considering the root node some correct hypotheses *may not* be generated when compared to analysing the entire code.

The penalty paid for analysing the whole code instead of just the root node is in computational time because of an increase in the number of false hypotheses generated. However computationally the approach is relatively inexpensive, and in practice the ranking of hypotheses produced by an indexing function could be used to effectively reduce the number of false hypotheses investigated.

By ranking the hypotheses according to non-formal information only the most interesting hypotheses need be investigated. The results obtained here show that in most cases these would be those most descriptive of the code's functionality and in only a very few cases would the hypothesis that corresponded to the actual function of the code not be generated. This suggests that such a top-down approach to redescribing source code could produce considerable benefits.

Given a system similar to Wills' Recogniser (Wills 1990) or Ning's PAT (Ning 1989; Harandi and Ning 1990) where high level design concepts are related to code via a grammar of some form, dramatic improvements could be made to performance by including non-formal analysis of the kind demonstrated here. If one considers each high level concept to broadly correspond to a class in a classification as developed here then even operating without any cutoff this would leave only approx 4.5 classes (as opposed to 22) to be investigated. (The ratio of 1:3.5 of the size of the sets corresponding to correct and incorrect indexings when no cutoff was applied was surprisingly constant between the design and test sets.) Clearly, in most cases, using non-formal information of this form to limit the search space would produce very significant improvements in performance.

7.8.2 Application to software reuse

There are two prospective uses for the approach to automatic indexing of source code investigated here. Firstly, such an approach could be used to index existing collections of software components, Secondly, such an approach could be used to identify potentially reusable components in existing source code.

The results here suggest that automatic indexing of software components using non-formal information is feasible. To use such an approach in practice would depend upon such code being reasonably well commented, however this is not necessarily a drawback since good commenting and good documentation is a prerequisite for software to be reusable.

The use of automatic indexing, based on an existing classification scheme, is unlikely to perform as well in component retrieval as manual indexing however it could present a cheap and labour saving alternative particularly where there are large volumes of code to be indexed. More research though would need to be done before undertaking such a task

Using non-formal information to locate potentially reusable components in existing source code is one possible way of identifying components with which to populate a reuse library. An approach similar to the one investigated here could be used to trawl through large volumes of existing code to identify code that performs a particular function. Candidates can then be ranked according to the likelihood of them implementing that function(s), ie being correctly indexed by a descriptor. Combined with other parameters to restrict the volume of code retrieved such as code size, specific reusable components could be identified within code at relatively little cost.

Furthermore, using non-formal information in the way described here is likely to rank code that is well commented much higher than code that is poorly commented which improves the likelihood of such code being suitable for reuse. In this application, using non-formal information does not suffer from the draw back of possible inaccuracy since potential components would not be entered into a component library without further (human) analysis. This should eliminate the possibility of a component being erroneously assigned a descriptor.

7.9 Extensions

There are two main ways in which the approach presented here can be extended to extract more detailed information from source code. Firstly, the ability of a thesaurus based approach to classifying code according to function could be extended by developing a more extensive thesaurus. This mainly applies to the use of non-formal information in redescribing source code. Secondly, the approach outlined here could be extended to index code according to the objects that a piece of code manipulates as well as the code's function. This mainly applies to the use of non-formal information as a means of automatically indexing source code for reuse.

7.9.1 Extending the thesaurus

The thesaurus that was developed as part of this project is more correctly described as a synonym dictionary. The reasons for using a very simple thesaurus in these experiments were twofold, firstly to accord with the kind of thesauri used in the classification schemes of Prieto-Diaz (Prieto-Diaz 1985) and of Wood (Wood 1987) and secondly to reduce the difficulty of evaluating the performance of the resulting indexing system.

The kind of use of non-formal information that Biggerstaff seems to be intending to implement in his Desire system (Biggerstaff et al. 1989) would appear to implicitly involve the implementation of a more detailed thesaurus than the one used here. This thesaurus would appear to have many links to both more general and more specific terms. These links are to be used to guide the search for conceptual abstractions within source code.

Implementing such a thesaurus as part of an attempt to use non-formal information in source code presents far more difficulties than the approach suggested here, although potentially it could lead to far more powerful analysis.

Using a thesaurus with links to related terms presents problems of controlling "noise". Generally, as the vocabulary of the thesaurus increases and the number of interconnections also increases there is likely to be an increasing problem with controlling ambiguous and spurious relationships between terms. Examples of these problems have already been given with the use of "deletes" as both a verb and a noun, and the different interpretations that can be applied to the term "add".

Controlling these problems generally involves being careful in the design of the thesaurus, and also through using context to disambiguate terms. The use of context could be particularly useful in linking together formal and non-formal forms of analyses. For example, resolving the context of the term "add" in source code may simply involve checking for the presence or absence of a relevant numerical calculation. Many other ambiguities may be resolvable in a similar way.

In general, the structure of source code and the limited vocabulary used within code (as opposed to the full range of natural language) would suggest that controlling the vocabulary used by a more complex thesaurus should be possible with a little care. The results of the analysis here, as compared to the results of similar experiments on general documents, further suggest that source code is likely to be amenable to this form of analysis.

7.9.2 Extending the analyses

Currently, only the use of non-formal information to identify the function of pieces of code has been pursued. Clearly, an important aspect of the nature of a piece of code is not only what it does, but also what it does ‘it’ to.

Any practical software classification scheme must have the ability to distinguish between components that manipulate different data structures or *entities*. With the increasing popularity of object oriented approaches to software development, particularly with respect to software reuse, it would be useful to be able to extend the approach described here to include information about the entities manipulated by a piece of code.

To do this is necessarily more complex than automatically indexing a piece of code according to the function. Prieto-Diaz's faceted classification scheme contains facets that are used to describe the objects manipulated by the code. However, unlike the function facet where most code can be adequately considered as performing a single function, Prieto-Diaz uses two facets to describe the objects which are manipulated by the code.

These two facets, *object* and *medium*, describe what *object(s)* is manipulated by the code, and what entity forms the *medium* in which this manipulation occurs. In this way, software components can be retrieved through queries of the form <function, object, medium>. Examples of such triples might be:

<add, character, string>
<search, string, file>
<copy, list, buffer>

The difficulty with attempting to automatically classify code by the entities it uses is illustrated in the above examples. In the first two examples, the entity <string> appears both as object and as medium. It is quite common for the same entity type to be able to appear as entries in different facets of a component description. As far as automatic indexing is concerned, this means that it is not enough to simply identify the entities used by a piece of code, it is also necessary to identify the *role* that these entities fill.

One possible approach to the above difficulty is to partially order the classes that used to describe the medium facet of the classification using a "component_of" relationship. This relationship would indicate when one such class can potentially be components of a different class.

A more complex, but more powerful approach could be based upon the use of templates to limit the number of combinations of descriptors that can be applied. Wood's component descriptor frame representations could be used to provide such a mechanism (*cf* 5.3).

Wood's (Wood 1987) approach to describing software components is based upon conceptual dependency theory (Schank 1975). Component descriptor frames (CDF's) describe a software component through the use of an *action* which describes the function of the code. Each action provides a skeletal CDF which is a frame like structure which specifies additional slots representing objects associated with the action. These slots need to be filled with appropriate objects to complete the CDF representation for a component. For a particular domain, a number of skeletal CDF's and objects can be defined.

Although Wood does not do so (though he does suggest such an extension) it would be possible to constrain the range of possible object fillers for each of the slots in a particular skeletal CDF. So for example, the range of values for the *destination* slot of the skeletal CDF for the action of *input-output* could be restricted to the subset of the possible *objects* that can function as input-output destinations. Similarly, constraints could be applied between slots of a particular CDF to represent information such as that contained in the "component_of" relation above.

Applying these constraints would limit the number of possible descriptions that could be applied to a particular piece of code. Implementing such an approach to provide automatic indexing of source code could involve developing some form of frame-based, predictive approach possibly similar to that used in natural language summarisers such as Frump (DeJong 1982) and IPP (Riesbeck 1982). These natural language based approaches, in common with the development of CDF's, were based on conceptual dependency theory.

Such an approach would involve using predictions as to the function performed by the code (possibly in the manner investigated as part of this thesis) to provide a range of hypotheses with which to describe the code, represented by skeletal CDF's. Attempts would then be made to attempt to find evidence in the source code to suggest particular fillers for the empty slots in the description.

At this point here, the complexity of the approach increases dramatically and rather than having a relatively simple and transparent approach to analysing source code we begin to encounter problems more traditionally associated with research in Artificial Intelligence and Natural Language Processing in particular. These problems would include difficulty with knowledge acquisition to supply the necessary domain knowledge, and technical difficulties surrounding solving problems which involve satisfying many competing constraints.

Also, the level of the description produced by such an analysis of source code makes clear the similarity between redescribing source code and the automatic indexing of code. This is because to effectively index source code, the description must reflect the 'content' or 'aboutness' of the code. In this sense the content of the code can be considered to be reified in a redescription of the code.

7.10 Limitations

The experimental results described above show how applications within reverse engineering and software reuse may benefit from performing some analysis of the non-formal information present in source code. In particular, the results suggest that tools for redescribing source code could achieve considerable improvements in performance due to the focusing of the search space achieved by analysing the non-formal information in code.

However, these are improvements in performance rather than significant improvements in the range and depth of code analysis that is possible. The analysis technique itself is also limited. Code that is very sparsely commented and poorly labelled will yield little information when its non-formal content is analysed. Code in which comments and labels are inaccurate or misleading will fail to produce any useful information via an analysis of its non-formal content.

The experimental work here can be seen as one more approach which aims to tackle the problem of the variety of ways that an individual 'design concept' can be implemented. The approach to this problem taken here is to attempt to correlate the occurrence of certain terms within source code with the function of the code. This has been achieved with some success, but the problem of the variation in the nature of source code still remains. This is demonstrated by the considerably higher minimum misclassification rate observed with the test set (0.26) as opposed to the design set (0.12). This is despite both samples of code being derived from the same system.

Ways of extending the analysis to improve this performance have been suggested. However, all the extensions and refinements to this analysis that have been suggested above involve a considerable increase in complexity. In general, these extensions all involve a move away from a straightforward statistical analysis of the code towards approaches with far more in common with the development of a 'knowledge based system'.

This trend towards increasing complexity of the systems required for redescribing source code is echoed by the redescription tools described in *chapter 3*. The penalty paid for this greater complexity is in increased development effort. This increased development effort is usually manifested by an increased proportion of time spent in performing domain analysis or in 'knowledge acquisition'. That is, less time (proportionately) will be spent on technical aspects of development and more time will be spent in domain specific tasks. This raises the question of how widely applicable such systems may be.

The similarity in results and conclusions from this work and from other attempts at developing redescription tools, despite widely disparate techniques, strongly suggests that these results are a consequence of the problem itself rather than the specific solution pursued. This hypothesis is to be examined in the final chapter.

7.11 Summary

This chapter has described the results of an experiment to automatically index pieces of source code by utilising the non-formal information present in the code. The results of this experiment demonstrate how applications in reverse engineering and software reuse could benefit from using a similar analysis of the non-formal content of source code.

Ways in which this kind of code analysis could be extended to provide more information about the nature of source code were outlined. However, the results of the experiment here show that non-formal analysis fails to significantly overcome the problems associated with the variety of ways that a particular 'design concept' or 'plan' can be implemented in source code.

Chapter 8
Future Directions

8.1 Introduction

Much of the research work on tools to automatically redescribe source code is based upon the assumption that the majority code can be adequately described by a set of stereotypical ‘design concepts’ or ‘plans’. A domain analysis can identify these ‘design concepts’ and the different ways that they can be implemented.

The major difficulty of this process, as perceived by the designers of such systems, is to find a way to overcome the large variety of ways in which a particular ‘concept’ or ‘plan’ can be implemented. Most research work has focussed on finding ways to combat this variety. This has involved the use of transformations based on mathematical properties of the code, the development of grammars which explicitly list all the variants, and the use of pattern matching techniques based upon distributed representations of domain knowledge.

The experimental work described in the previous three chapters has similarly aimed to attack this problem by analysing non-formal information in source code. Whilst partially successful, the results of this experiment fail to indicate that such an approach will significantly improve the ability of redescription tools to identify different implementations of ‘design concepts’. This failure, and the similar failure of other redescription tools, motivates a closer look at the assumptions underlying attempts to redescribe source code.

Any approach to automatically redescribe code uses some form of fixed ‘classification’, this can be considered as the set of higher level descriptions or structures which are to form the basis of the new description (this is true even for transformational approaches which search for patterns in the code that can be transformed to a more abstract representation).

If we apply some of the arguments against the existence of a neutral and objective universe of given objects and classes developed in *chapter 4* to the use of a classification for redescribing source code, the problem of the variability of source code is recast. Rather than consider the problem of accurately classifying source code as a problem of overcoming the diversity of implementation, we need to consider the suitability of the classification itself for describing the code.

Obtaining some form of higher level description of existing source code can be seen as a problem of capturing the 'aboutness' of a piece of code. The notion of 'aboutness' refers to the intrinsic subject of a document. This is assumed to be to independent of the temporary use to which an individual user may put the document (for a fuller discussion of 'aboutness' see (Beghtol 1986)).

Once produced this higher level description of code can be considered as a text in its own right. This text is obviously related to the original source code, but what is the nature of this relationship? Beghtol (Beghtol 1986), drawing upon the textual semantics of Van Dijk, notes that the aboutness of a document depends not only on the contents of the original text, but also on four extra-textual elements. These are given as:

1. *The cultural tradition* - this is broadly equatable to Eco's notion of cultural units (*cf* 4.4)

2. *The reality of the moment* - this includes the reason why the reader is reading the text.

3. *The original author of the text* - what was the intention behind the writing of the text?

4. *The percipient (reader) of the text* - the person attempting to redescribe the text (source code).

This list of extra-textual features which affect the 'aboutness' of a document (ie source code) suggests that the source code itself is far from the sole determinant of what qualifies as a good description of the code. The assumptions which underly attempts to automatically redescribe source code insist that this aboutness relation can be approximated by a function of the source code alone. This ignores the variability that can be associated with these extra-textual features. The contribution of the cultural tradition is especially important as has already been emphasised in *chapter 4.*

In certain areas, for example simple programming constructs and data types, it is reasonable to expect some success in attempting to identify (classify) these structures within code. With such fundamental and well established structures, the redescription problem may well reduce to a problem of formalising the cultural conventions which account for the implementation and interpretation of these structures. As current research has demonstrated, this is a difficult enough undertaking in itself.

However, identifying such simple structures in source code is far from automatically producing designs and specifications from code. If we wish our redescriptions of code to be at a high enough level of abstraction to enable us to re-establish the link between code and the application, then the extra-textual elements influencing the aboutness of a document presents more serious obstacles to the automation of this process. We may find that our understanding of the system will be affected by the way that the system is currently used, we may wish to document the reasons as to *why* a particular piece of code was implemented the way it is, and we will probably find that different people conceptualise the workings of the same system in different ways.

Reverse engineering can be viewed as the *construction* of a description of an existing system, but this description is not constructed exclusively from fixed, given, structures. The importance of the link with the application ensures that, in a process similar to that which occurs in the early stages of forward engineering, reverse engineering must construct not only a description of the system but more importantly *a language* with which to describe the system (Holmqvist and Andersen 1991, Turner 1987).

Any reverse engineering tool must be sufficiently flexible to allow this construction of a description language. There must be flexibility in the tool which will allow the extra-textual influences on the 'aboutness' of the original code to be incorporated into the new description. This is an alternative way of noting the *embeddedness* of software systems. That is the way that a software system acts on, and is acted on by its environment. This environment includes not only software and hardware, but also the system's users and the organisation within which the system works.

Any description of a system must take into account this embeddedness. In redescribing source code this means that tools to support the redescription of code must be able to incorporate concerns originating from the system's use into the new description. Any tool which overly constrains the range of descriptions which can be applied to code is likely to inhibit, rather than facilitate, the reverse engineering process. This view also has implications for the practice of software reuse since it suggests that there are limits to the effectiveness of classifying software components in a manner that considers them independently of their use.

The inherent originality and embeddedness of software systems would seem to limit the usefulness of tools which aim to automatically redescribe source code. Tools to redescribe source code will inevitably remain as tools, and are unlikely to produce dramatic improvements to the practicality of reverse engineering systems. Given that this is the case, then it is necessary to give more consideration to the way these tools might actually be used than has been done at present.

8.2 Maintenance in the Large

The tools described in *chapter 3* have all been envisaged as being used to analyse small parts of computer systems, either individual modules or procedures. The process of analysing such small parts of a system as compared to performing maintenance activities *on* the system are quite distinct.

In practice, the source code is only one component of a large system. The source code is one document amongst the many that are associated with a piece of software. Specifications, designs, technical manuals, user manuals etc. are all important parts of the system. Further, all such systems are embedded in the real world, and so the way that the system is used and the effect that the system has on users is also of importance in maintenance.

Systems development depends upon using an implicit set of values to narrow down the focus of the development activity. Similarly, the way the system is used and the way system features are communicated by users constitutes a language which is central in determining the way the system is perceived (Holmqvist and Andersen 1991). For successful large scale maintenance, both these implicit values used and the meaning given to the system by its users need to be uncovered.

Identifying these features associated with a system is not possible simply by analysing source code. Thus the process of reverse engineering, and similar maintenance activities, needs to consider the operation of the system *in situ*, considering the systems situatedness both in terms of its original development and the perceived need for the system at that time. Much of this can involve studying the way that users give meaning to the operation of the system by the interpretation of its actions (see for example Boland 1991).

Analysing a system in this way presents far more difficulties and is perhaps far more central to successful maintenance than the process of redescribing source code. It is this kind of shift in emphasis that motivated the view of source code as text, and considering the semiotics of the text as a basis for analysis.

This view of systems development and maintenance is consistent with a *process oriented* view as opposed to a *product oriented* view (Floyd 1988). The *product-oriented perspective* regards software as a stand alone product. Such a product (which includes programs and documentations) is considered to be independent of use. The *process-oriented perspective* considers software to be embedded within a constantly evolving world. The software is viewed in its connection to human learning, communication and work.

If the process oriented perspective is fully adopted the idea of maintenance as a separate phase of the software lifecycle has to be rejected. When a software system is considered within the context in which it is used, the notion of software maintenance is replaced with seeing a software system present in a constant cycle of change. This is a cycle of change in which there is reciprocal action between the system and its environment, where the interaction between the system and its environment is tailored by the users of the system so that the system can support the needs of the users work processes. Such a viewpoint suggests that future research in software maintenance should place more emphasis on finding ways of analysing the nature of this relationship between system and environment rather than on tools which analyse the system in isolation.

8.3 Future Directions

The adoption of the process-oriented viewpoint is dependent upon the rejection of a simplistic view of meaning, and in particular semantics, as being concerned with ways of referring to an objective universe of fixed and discrete objects. *Chapter 4* was partly written in an attempt to refute this view and so pave the way for a conception of meaning which emphasises the role of culture, convention and communication. Whilst there have been many generalised discussions about various conceptions of meaning, it seemed useful to apply these arguments directly to the specific case in question, namely source code, and so demonstrate the practical differences between these two viewpoints.

On a wider basis, this perspective requires that research work notes the situatedness of both the software system *and* the observer. This requires the adoption of research techniques derived from the realm of the social sciences, fields such as cultural anthropology and linguistics.

Key issues to be addressed in such studies would be the way in which tools and methodologies for software development (including maintenance) contribute to the way these activities are understood and conceptualised. Of particular relevance to this thesis would be an investigation into the way tools for redescribing source code may be used in practice, and the effect that this may have on software maintenance and the way maintenance activities are perceived.

Of particular use in this kind of study would be a semiotic framework which recognises that when people act in an organisation (say in performing maintenance) they do so through a variety of symbol systems, such as language, technology and process. Semiotics, and in particular semantics, can be used as a basis for studying many aspects of a software system. As well as considering texts and verbal communication as sign systems, actions (both human and computer) can also be interpreted as signs to be analysed. Similarly tools, such as those for redescribing source code, and methodologies can also be considered as constituents of sign systems (Stamper 1987).

The nature of these sign systems will have effect on the process of maintenance, and on the interpretation of the eventual outcome of maintenance. An analysis of these sign systems may yield useful insights into issues surrounding software development. These kind of studies focus on the way that sign systems enable users to give meaning to system outputs, including documentation. For examples of this approach to research in information systems see (Boland 1991, Truex and Klein 1990, Boland and Hirshheim 1987, Hirchheim and Klein 1986).

8.4 Summary

This thesis has tried to present a view that the field of software maintenance could benefit from considering source code as a text. This involves considering the role source code plays in providing a channel of communication from human to human as opposed to the more restricted view of its role in human-computer communication.

To accomplish this aim, this thesis has described one particular task which a number of researches from different backgrounds are attempting to tackle - that of producing high level descriptions of existing source code. This task has been called here the task of *redescribing source code* and has been related to research within the areas of reverse engineering and software reuse.

Some of the difficulties associated with viewing source code merely as a means of human-computer communication have been highlighted in *chapter 4.* The intention being to point to deficiencies of some existing approaches to redescribing source and hence show that there is a need to consider non-formal information in source code analysis. Using non-formal information in the analysis of source code is characteristic of considering textual properties of the code.

This view of source code is used to provide a new perspective on some of the approaches to redescribing source code by relating these approaches to similar work that aims to analyse natural language text. This also leads to suggestions for new methods for analysing source code.

In addition to presenting a theoretical argument for considering source code as text, an empirical investigation designed to support this view was undertaken. This investigation attempted to use non-formal information in source code as a basis for automatically indexing the code with descriptors that correspond to the function of the code.

This practical work was carried out on commercially developed code. The results obtained demonstrated that information about the nature of source code can be obtained from considering non-formal features of source code. This information could be made use of by tools to redescribe source code and by tools to locate and index software components for reuse.

However, the results of the experimental work suggests that the use of non-formal information by source code redescription tools will fail to significantly reduce the problems associated with the wide variety of ways that individual 'design concepts' or 'plans' can be implemented. This suggests that some of the assumptions underlying attempts to develop tools to redescribe source code may be inappropriate.

An application of the line of argument used in *chapter 4* to critique current approaches to redescribing source code to the process of redescribing source code concludes that most attempts to develop redescription tools are overly constricting in their conception of software maintenance and reverse engineering. Following this line of reasoning suggests that future research should consider in far more detail the way that redescription tools might be used in practice. In particular, this research should be aware of the wider environment in which system use and maintenance occurs and not consider software systems and source code as existing independently of this environment.

References

James P. Ambras, Lucy M. Berlin, Mark L. Chiarelli, Alan L. Foster, Vikki O'Day, and Randolph N. Splitter, ''MicroScope: An Integrated Program Analysis Tool,'' *Hewlett-Packard Journal*, August 1988.

Victor R. Basili, ''Reusing Existing Software,'' Report UMIACS-TR-88-72, Institute for Advanced Computer Studies, University of Maryland, USA, Oct 1988.

C. Beghtol, ''Bibliographic Classification Theory and Text Linguistics: Aboutness, Intertextuality and the Cognitive Act of Classifying Documents,'' *Journal of Documentation*, vol. 42, no. 2, pp. 84-113, June 1986.

James O. Berger, *Statistical Decision Theory and Bayesian Analysis,* Springer-Verlag, 1980.

Ted J. Biggerstaff, J.C. Hoskins, and D. Webster, ''DESIRE: A System for Design Recovery,'' MCC Technical Report STP-081-89, May 1989.

Ted J. Biggerstaff and Alan J. Perlis Eds., *Software Reusability: Concepts and Models,* 1, ACM Press, 1989.

Ted J. Biggerstaff and Alan J. Perlis Eds., *Software Reusability: Applications and Experience,* 2, ACM Press, 1989.

Ted J. Biggerstaff, ''Design Recovery for Maintenance and Reuse,'' *Computer*, July 1989.

R.J. Boland and R.A. Hirschheim (Eds), *Critical Issues in Information Systems Research,* Wiley, 1987.

R.J. Boland, ''Information System Use as a Hermeneutic Process,'' Weatherhead School of Management, Cleveland, Ohio, 1991.

C. Boldyreff, ''Automating the Analysis of Source Code to Support Reuse: A Survey of Relevant Work and Available Tools,'' Working Paper, Brunel University, Uxbridge, 1989.

Cornelia Boldyreff and Jian Zhang, ''From Recursion Extraction to Automated Commenting,'' in *Reuse, Maintenance and Reverse Engineering of Software: Current Practice and New Directions, Unicom Seminar*, 29 Nov-1 Dec 1989..

C. Boldyreff, P. Elzer, P.A.V. Hall, U. Kaaber, J. Keilmann, and J. Witt, ''PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software ,'' in *SE 90, Proceedings of Software Engineering 90*, Brighton, July 1990.

C. Boldyreff and H. Albrechtsen, ''Software Classification - A Brief Review of Approaches,'' Working Paper, Brunel University, Uxbridge, Feb 1990.

Ruven Brooks, ''Towards a Theory of the Comprehension of Computer Programs,'' *International Journal of Man-Machine Studies*, vol. 18, 1983.

D. Brotsky, ''An Algorithm for Parsing Flow Graphs,'' Tech. Report 704 (M.S. Thesis), Artificial Intelligence Lab. MIT., Cambridge, MA, 1984.

A.J. Brown, ''Specifications and Reverse Engineering,'' *Journal of Software*

Maintenance, To be published 1993.

B. Carre and D.L. Clutterbuck, "The Verification of low level code," *Software Engineering Journal*, May 1988.

Yih-Farn Chen and C.V. Ramamoorthy, "The C Information Abstractor," *IEEE*, 1986.

Yih-Farn Chen, Michael Y. Nishimoto, and C.V. Ramamoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, March 1990.

E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan 1990.

T.A. Corbi, "Program Understanding: Challenge for the 1990's," *IBM Systems Journal*, vol. 28, no. 2, Feb 1989.

S.P. Davies, "The Nature and Development of Programming Plans," *International Journal of Man Machine Studies*, vol. 32, 1990.

Gerald DeJong, "An Overview of the Frump System," in *Strategies for Natural Language Processing*, ed. W.G.Lehnert and M.H. Ringle, L. Erlbaum, 1982.

F. Detienne and E. Soloway, "An Empirically-Derived Control Structure for the Process of Program Understanding," *Int. J. Man-Machine Studies*, vol. 33, pp. 323-342, 1990.

E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

Umberto Eco, *A Theory of Semiotics,* Indiana University Press, 1976.

Umberto Eco, *The Role of the Reader,* Hutchinson, 1979.

Norman E. Fenton and Agnes A. Kaposi, ''Metrics and software structure,'' *Information and software technology,* vol. 9, no. 6, July/Aug 1987.

J.H. Fetzer, ''Program Verification: The Very Idea,'' *Communications of the ACM,* vol. 31, no. 9, September 1988.

R.K. Fjeldstad and W.T. Hamlen, ''Application program maintenance study - report to our respondents.,'' in *Tutorial on Software Maintenance,* ed. N Zvegintzov, IEEE Computer Society Press, 1983.

C. Floyd, ''Outline of a Paradigm Change in Software Engineering,'' *Software Engineering Notes,* vol. 13, no. 2, pp. 25-39, April 1988.

J.R. Foster, A.E.P. Jolly, and M.T. Norris, ''An Overview of Software Maintenance,'' *British Telecom Technology Journal,* vol. 7, no. 4, Oct 1989.

W.B. Frakes and B.A. Nejmeh, ''Software Reuse through Information Retrieval,'' *SIGIR Forum,* vol. 21, no. 1-2, 1986-1987.

W. Frey, U. Reyle, and C. Rohrer, ''Automatic Construction of a Knowledge Base by Analysing Texts in Natural Language,'' in *Proceedings 8th International Joint Conference on Artificial Intelligence IJCAI 83,* pp. 727-729, Karlsruhe, 1983.

Norbert Fuhr, ''Models for Retrieval with Probabilistic Indexing,'' *Information Processing and Management,* vol. 25, no. 1, pp. 55-72, 1989.

Norbert Fuhr, ''Optimum Polynomial Retrieval Functions,'' in *Research and Development in Information Retrieval*, ed. C.J. van Rijsbergen, ACM Press, 1989.

Norbert Fuhr and Chris Buckley, ''Probabilistic Indexing from Relevance Feedback Data,'' in *Proceedings of the 13th International Conference on Research and Development in Information Retrieval, Brussels, 5-7 September 1990*, ed. Jean-Luc Vidick, Pub. Presses Universitaires De Bruxelles, 1990.

E.S. Garnett and J.A. Mariani, ''Software Reclamation,'' *Software Engineering Journal*, vol. 5, no. 3, pp. 185-91, May 1990.

J.D. Gibbons, *Nonparametric Statistical Inference,* Marcel Dekker, Inc., 1985.

D.J. Gilmore and T.R.G. Green, ''Programming Plans and Programming Expertise,'' *The Quarterly Journal of Experimental Psychology*, vol. 40A, no. 3, 1988.

Raymonde Guindon, ''Knowledge exploited by experts during software system design,'' *Int. J. Man-Machine Studies*, vol. 33, pp. 279-304, 1990.

M.T. Harandi and J.Q. Ning, ''Knowledge Based Program Analysis,'' *IEEE Software*, Jan 1990.

John Hartman, ''Understanding Natural Programs Using Proper Decomposition,'' in *13th I.C.S.E.*, May 1991.

John Haugeland, *Artificial Intelligence: The Very Idea,* MIT Press, 1985.

P.A. Haulser, M.G. Pleszkoch, R.C. Linger, and A.R. Hevner, ''Using

Function Abstraction to Understand Program Behaviour,'' *IEEE Software*, Jan 1990.

H.S. Heaps, *Information Retrieval,* Academic Press, 1978.

R. Hirchheim and H. Klein, ''The Emergence of Pluralism in Information Systems Development,'' RDP 86/15, Oxford *Institute of Information* Management, Templeton College, Oxford, 1986.

C.A.R. Hoare, ''An axiomatic basis for computer programming,'' *Communications of the ACM*, vol. 12, 1969.

B. Holmqvist and P.B. Andersen, ''Language, Perspectives and Design,'' in *Design at Work*, ed. Greenbaum and Kyng, Earlbaum, 1991.

James W. Hooper and Rowena O. Chester, *Software Reuse: Guidelines and Methods,* Plenum Press, 1991.

W.L. Johnson and E. Soloway, ''Proust: Knowledge-based program understanding,'' in *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984.

V. Karakostas, *Automated Business Knowledge Acquisition from Programs,* Department of Computation, UMIST, Privately obtained 1991.

Gerhard Knorz, ''A Decision Theory Approach to Automatic Indexing,'' in *Research and Development in Information Retrieval*, ed. Gerard Salton and Hans-Jochen Schneider, Lecture Notes in Computer Science, Springer-Verlag, 1982.

Bogdan Korel, ''PELAS - Program Error Locating Assistant System,'' *IEEE Transactions on Software Engineering*, vol. SE-14, no. 9, Sept 1988.

A. Korzybski, *Science and Sanity,* The International Non-Aristotelian Library, 1958.

Wojtek Kozaczynski and Jim Q. Ning, ''SRE: A Knowledge-based Environment for Large Scale Software Re-engineering Activities,'' in *Proceedings of the International Conference on Software Engineering*, IEEE CS Press, 1989.

Imre Lakatos, *Proofs and Refutations,* Cambridge University Press, 1976.

F.W. Lancaster, *Information Retrieval Systems - Characteristics, Testing, and Evaluation,* John Wiley and Sons, 1979.

K. Lano and P.T. Breuer, ''From Programs to Z Specifications,'' in *Z User Workshop, Oxford 1989,* ed. J. E. Nicholls, Springer-Verlag, 1990.

Adam, Anne and Jean-Piere Laurent, ''LAURA. A System to Debug Student Programs,'' *Artificial Intelligence*, vol. 15, 1980.

M.M. Lehman, ''Programs, Life Cycles, and Laws of Program Evolution,'' *Proceedings of the IEEE*, vol. 68, no. 9, 1980.

M.M. Lehman and L.A. Belady, *Program Evolution,* Academic Press, 1985.

Philip Leith, *Formalism in AI and Computer Science,* Ellis Horwood, 1990.

D. B. Lenat, ''AM: An Artificial Intelligence Approach to Discovery in

Mathematics,'' in *Knowledge-Based Systems in Artificial Intelligence*, McGraw Hill, 1982.

Stanley Letovsky, ''Cognitive Processes in Program Understanding,'' in *Empirical Studies of Programmers*, ed. S. Iyengar, Ablex, Norwood NJ, 1986.

Stanley Letovsky, ''Plan Analysis of Programs,'' YALEU/CSD/RR 662, Yale University, December 1988.

Stephen C. Levinson, *Pragmatics*, Cambridge University Press, 1983.

Y.S. Maarek and F.Z. Smadja, ''Full Text Indexing Based on Lexical Relations,'' in *SIGIR 89*, ed. C.J. van Rijsbergen, pp. 198-206, ACM Press, June 1989.

J.Q. Ning, ''A Knowledge Based Approach to Automatic Program Analysis,'' Doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois., 1989.

F. Nishida, S. Takamatsu, T. Tani, and H. Kusaka, ''Text Analysis and Knowledge Extraction,'' in *Proceedings 11th International Conference on Computational Linguistics COLING 1986*, pp. 241-243, Bonn, 1986.

Paul W. Oman and Curtis R. Cook, ''Typographic Style is More than Cosmetic,'' *Communications of the ACM*, vol. 33, no. 5, May 1990.

Roger S. Pressman, *Software Engineering: A Practitioners Approach,* McGraw-Hill, 1987.

R. Prieto-Diaz, ''A Software Classification Scheme,'' Doctoral Dissertation, Department of Information and Computer Science, University of California, 1985.

R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, January 1987.

R. Prieto-Diaz, "Domain Analysis: An Introduction," *ACM Software Engineering Notes*, April 1990.

R.G. Reynolds, J.I. Maletic, and S.E. Porvin, "PM: A System to Support the Automatic Acquisition of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, Sept 1990.

Charles Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proc of the 7th International Conference on Artificial Intelligence*, vol. 2, 1981.

Charles Rich, "The Layered Architecture of a System for Reasoning about Programs," in *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985.

Charles Rich and Richard C. Waters, "The Programmer's Apprentice: A Research Overview," *Computer*, Nov 1988.

Charles Rich and Richard C. Waters, *The Programmers Apprentice,* ACM Press, 1990.

Paul Ricoeur, *Hermeneutics and the Human Sciences,* Cambridge University Press, 1981.

C.K. Riesbeck, "Realistic Language Comprehension," in *Strategies for Natural Language Processing*, ed. W.G. Lehnert and M.H. Ringle, L. Erlbaum, 1982.

C.J. van Rijsbergen, *Information Retrieval,* Butterworths, 1979.

R.S. Rist, "Plans in Programming: Definition, Demonstration and Development," in *Empirical Studies of Programmers*, ed. E. Soloway and S. Iyengar, Ablex, Norwood NJ, 1986.

S. Rugaber, S.B. Ornburn, and R.J. LeBlanc Jr, "Recognising Design Decisions in Programs," *IEEE Software*, Jan 1990.

D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing,* MIT Press, 1986.

Gerard Salton and Michael J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, 1983.

Robert J. Schalkoff, *Pattern Recognition: Statistical, Structural and Neural Approaches,* John Wiley and Sons Inc., 1992.

R.C. Schank, *Conceptual Information Processing,* North-Holland/American Elsevier, 1975.

R.C. Schank and R. Abelson, *Scripts, Plans, Goals and Understanding,* Erlbaum, 1977.

Peter G. Selfridge, "Integrating Code Knowledge with a Software Information System," in *Proceedings 5th Annual Knowledge-Based Software*

Assistant Conference, Syracruse NY, Sept 24-28 1990.

B. Sheil, ''The Psychological Study of Programming,'' *Computing Surveys*, no. 13, 1981.

Raymond M. Smullyan, *Theory of Formal Systems,* Princeton University Press, 1961.

Harry M. Sneed, ''SOFTDOC Static Analyser,'' System Documentation, Vers. 5, Munich, 1985.

Harry M. Sneed and Andras Merey, ''Automated software quality assurance,'' *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, Sept 1985.

Harry M. Sneed and Gabor Jandrasics, ''Software recycling,'' in *Proceedings from the Conference on Software Maintenance, Austin, Texas.*, Sept 21-24 1987.

E. Soloway and K. Ehrlich, ''An Empirical Investigation of the Tacit Plan Knowledge in Programming,'' in *Human Factors in Computer Systems*, ed. J.C. Thomas and M.L. Schneider, Ablex, 1984a.

E. Soloway and K. Ehrlich, ''Empirical Studies of Programming Knowledge,'' *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984b.

E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, ''What Do Novices Know About Programming?,'' in *Directions in Human Computer Interaction*, ed. B. Shneiderman, Ablex, 1982.

J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine,* pp. 42-51, Addison-Wesley, 1984.

K. Sparck-Jones, *Automatic Keyword Classification,* Butterworths, 1971.

Ronald Stamper, "Semantics," in *Critical Issues in Information Systems Research,* ed. R.J. Boland and R.A. Hirschheim, Wiley, 1987.

Stan Szpakowicz, "Semi-automatic acquisition of conceptual structure from technical texts," *International Journal of Man-Machine Studies,* vol. 33, 1990. 385-397

Ted Tenny, "Program Readability: Procedures Versus Comments," *IEEE Transactions on Software Engineering,* vol. 14, no. 9, September 1988.

P. Tibbetts, "Representation and the realist-constructivist controversy," in *Representation in Scientific Practice,* ed. S. Woolgar, pp. 69-84, Kluwer Academic Publishers, 1988.

D. Truex and H. Klein, *The Rejection of Structure as a Basis for Information System Design,* Presented at COSCIS, 1990.

Jon A. Turner, "Understanding the Elements of System Design," in *Critical Issues in Information Systems Research,* ed. R.J. Boland and R.A. Hirschheim, Wiley, 1987.

W.M. Turski and T.S.E. Maibaum, *The Specification of Computer Programs,* Addison-Wesley, 1987.

Martin Ward, F.W. Callis, and M. Munro, *The Use of Transformations in The*

Maintainer's Assistant, Centre for Software Maintenance, University of Durham, Privately obtained, Dec 1989.

Martin Ward, *The Formal Derivation of Specifications from Code,* Centre for Software Maintenance, University of Durham, Computer Science Technical Report, Privately obtained, Dec 1989.

A. Wasserman, ''Information System Design Methodology,'' in *Software Design Techniques*, ed. A. Wasserman, IEEE Computer Society Press, 1983.

Richard C. Waters, ''A Method For Analysing Loop Programs,'' *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, May 1979.

P. Wegner, ''Capital-Intensive Software Technology,'' *IEEE Software*, vol. 1, no. 3, July 1984.

Mark Weiser, ''Program Slicing,'' *IEEE Transactions on Software Engineering*, vol. 10, no. 4, July 1984.

H. Wertz, *Automatic Correction and Improvement of Programs,* Ellis Horwood series in AI, 1987.

B.L. Whorf, *Language, Thought and Reality,* MIT Press, 1956.

S. Wiedenbeck, ''Processes in Program Comprehension,'' in *Empirical Studies of Programmers*, ed. S Iyengar, Ablex, Norwood NJ, 1986.

L.M. Wills, ''Automated Program Recognition,'' Tech. Report 904 (M.S. Thesis), Artificial Intelligence Lab. MIT., 1986.

L.M. Wills and C. Rich, ''Recognising a Programs Design: A Graph Parsing Approach.,'' *IEEE Software*, Jan 1990.

L.M. Wills, ''Automated Program Recognition: A Feasibility Demonstration,'' *Artificial Intelligence*, no. 45, pp. 113-171, 1990.

T. Winograd and F.C. Flores, *Understanding Computers and Cognition,* Ablex, 1986.

N. Wirth, ''Program Development through Stepwise Refinement,'' *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, April 1971.

Ludwig Wittgenstein, *Remarks on the Foundations of Mathematics,* Basil Blackwell, 1978.

Murray Wood, ''Component Data Frames: A Representation to Support the Storage and Retrieval of Reusable Software Components,'' PhD. Thesis, University of Strathclyde, 1987.

Murray Wood and Ian Sommerville, ''An Information Retrieval System for Software Components,'' *Software Engineering Journal*, Sept 1988..

C.T. Yu and G. Salton, ''Effective Information Retrieval using Term Accuracy,'' *Communications of the ACM*, vol. 20, no. 3, pp. 135-142, March 1977.

Appendix 1
Thesaurus Classes

abend = {abend abort quit}

activate = {activate initiate start}

add = {add include insert push}

allocate = {allocate allot assign}

backout = {backout undo}

commit = {commit consign}

create = {assemble build create generate make}

delete = {cancel delete erase purge remove strip}

free = {free discharge release}

get = {acquire choose extract get obtain}

initialise = {clear initialise}

process = {handle process}

quiesce = {quiesce shutdown silence sleep}

reply = {answer reply respond}

search = {find locate look match scan search traverse}

select = {choose decide extract pick select}

send = {communicate inform send tell}

set = {assign define reset set}

sort = {arrange order organise rank sort}

terminate = {cease complete finish stop terminate}

update = {change insert modify update}

validate = {check confirm validate verify}

wait = {hold remain wait}

Appendix 2
Lexicon Entries

acquire = acquir*

add = add | adds | added

answer = answer*

allocate = alloc*

arrange = arrange*

build = build | builds | built | bld

change = chang*

commit = commit*

communicate = communicat*

confirm = confirm*

create = create* | creation

delete = del | delete | deleted

free = free | freed | frees | freemain*

find = find | finds | found | fnd

get = get | gets | getmain*

handle = handl*

include = include*

inform = inform | informed

initialise = init | initiali*

insert = ins*rt*

make = make | made | mk

process = process*

quiesce = quiesc*

release = release | released

reply = reply | replied

respond = respond*

scan = scan*

send = send*

search = search* | srch

sleep = sleep*

sort = sort | sorted

tell = tell | told

terminate = terminat*

update = updat* | upd

verify = verif*