## *Documentation for software maintenance and there documentation of existing systems*

Nigel Thomas Fletton

# Documentation for Software Maintenance and the Redocumentation of Existing Systems

Nigel Thomas Fletton

Thesis submitted for the degree of
Master of Science

University of Durham
School of Engineering and Applied Science
Computer Science

30th September, 1988



Nigel Thomas Fletton

# Documentation for Software Maintenance and the Redocumentation of Existing Systems


## Abstract

The importance of software documentation in maintenance work is widely acknowledged by those involved in the work. However, many new software projects are still being produced with documentation that is inadequate for efficient support of the product following development. When a product enters the maintenance phase of its life-cycle, the need for quality documentation increases dramatically as it is common for the maintenance team to be composed of personnel who were not involved in the products development. This thesis surveys the tools available for supporting the production of software documentation and then proposes a tool, based on hypertext technology, that will enable maintenance programmers to efficiently create documentation about systems they are working on, where the existing documentation is unsatisfactory.

**Acknowledgements**

The author would like to thank the Director of British Telecom Research Laboratories for the opportunity to study for this M.Sc. Thanks are also due to the many friends and colleagues in the Systems and Software Engineering Division and particularly to the past and present members of the Software Engineering Applications Group who have contributed to the DOCMAN system on which this research is based.

Finally, I would like to thank my supervisor, Malcolm Munro, for his guidance over the past year.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The term 'Software Crisis' has been used to describe the problems that have been encountered in producing large software products. It has been discussed for many years in the software engineering literature, but still there appears to be no solution available as the symptoms are still very much apparent. In the future, perhaps areas such as formal methods, CASE tools, rapid prototyping and IPSEs will offer some solution. At the moments these areas are in their infancy and have not received wide spread acceptance. Even when they do, it will be some time before their effectiveness at reducing the symptoms of the software crisis can be established.

The crisis comes about from the rapidly decreasing hardware costs and increasing hardware capacity. Boehm[9] first presented the hardware/software cost trends diagram that predicted a dramatic rise in software costs relative to hardware cost at a time when spending more on software than hardware was difficult for many to perceive. Exploiting this increased capacity has lead to an increase in software complexity and cost which in turn has highlighted the problems associated with managing large software projects.

Personnel and skill shortages have also been a major contributor in ensuring that the software crisis continues.



## 1.1 Software Maintenance

The phrase 'software maintenance' has been defined in varying ways by many people. Any disagreement usually centres on the number of activities that the term encompasses. I shall use a broad definition given by Foster[32]:

> Software maintenance is the set of activities associated with keeping operational software in tune with the requirements of its users and operators, and of all other people and systems with which the operational system interacts.

Software maintenance activities are commonly classified into four areas based on the categories first offered by Swanson[61]. These areas are: corrective maintenance, adaptive maintenance, perfective maintenance and preventive maintenance.

Within the software industry there is a growing awareness of the significance of software maintenance as an activity that deserves specific attention. This awareness can be attributed to a small group of academic and industrial gurus who over the last 10–15 years have been debating software maintenance and its associated problems. This debate has resulted in the recognition of software maintenance as an important area by many in the software industry. However, there are still large organisations that have not identified software maintenance as a problem. Those that have may well have been influenced by the results of surveys published, mainly in the US DP sector, which have shown that 30–80% of software expenditure is spent on existing software[66]. There are no reasons to doubt that these figures equally apply to the British software industry and to the maintenance of real-time software systems.

With such a large proportion of the total software expenditure being spent on software maintenance, this area has the greatest potential of any in the software life cycle for reducing overall system costs. The direction of money into the maintenance of existing systems has caused new developments to be postponed due to lack of financial and personnel resources. Any freeing of money from software maintenance, by increasing maintenance programmer's productivity or better software maintenance management, would help reduce the development backlog created by these resource shortages.

Most research into software engineering has been centred on improving techniques and methods of the early parts of the software life-cycle. These are the requirements, specification, design and implementation phases. This work has been valuable in improving the quality of new developments and has undoubtedly helped reduce the maintenance burden. Regrettably, it has not addressed the problems of the large body of programmers who are working on existing systems, designed before the wide spread use of modern programming practices.

There are two approaches to alleviating the maintenance problem. Firstly, as mentioned above, the development process can be improved to reduce the need for maintenance. The most recent research in this area includes programming methodologies and software development environments. Secondly, the maintenance problem can be approached directly and methods of reducing the maintenance overhead of software systems can be identified. This approach is effective for both new and old systems, developed with or without modern programming practices. Both approaches are worthy of attention, but the latter approach has not been given the attention it deserves considering its potential for direct financial gains.

Major problems faced when maintaining old software are:

- Much of existing software is unstructured and is written in languages that do not easily support structured programming techniques. Unfortunately the momentum of these languages will ensure their continued use for many years to come[66].
- Generally maintenance programmers have not been involved in a products development prior to maintaining it. This unfamiliarity causes programmers to be heavily reliant on the support documentation.
- The software documentation is often nonexistent, incomplete or out of date. Where it does exist, it usually consists of an unmanageable set of unstructured papers that are difficult to access and impossible to maintain.

## 1.2 Objectives

There are many areas in which the software maintenance activity can be improved. I shall not list them all here as they have already been identified by many authors. The area of specific interest in this research is the documentation of program source code and centres on two main questions: what form of source code documentation is the most useful to maintenance programmers and how should this information be presented? Documentation has already been identified as a major contributor to the high cost of software maintenance[46]. A survey by Chapin[21] of personnel close to software maintenance work showed that they perceived poor documentation as the biggest problem in software maintenance work.

The objectives for the research described in this thesis were:

1. Investigate the categories and problems of software documentation.
2. Survey source code specific documentation tools.
3. Investigate the application of hypertext technology to redocumenting software systems.
4. Develop a strategy for redocumenting source code during software maintenance.
5. Establish requirements for a redocumentation system.

## 1.3 Overview of Thesis

Chapter 2 discusses source code documentation in general: including its properties and its components. A survey of the current state of art in tools specifically targeted at software documentation is contained in Chapter 3. Redocumentation is discussed in Chapter 4 and the high level requirements of a system for redocumenting programs during software maintenance are given. These lead to the work on developing ideas on using hypertext for the framework of a redocumentation system in Chapter 5 and the development of a prototype hypertext system for browsing and documenting source code in Chapter 6. Chapter 7 suggests paths for further research and development based on the ideas presented in this thesis.

Finally, Chapter 8 contains the conclusions of this research.

# Chapter 2

# Software Documentation

## Introduction

Software documentation is a means of communicating knowledge about a program in an alternative form or a more abstract and easily understood form than that available from the source code itself. The software documentation produced during the development of a system is more than a description of a program. It is a set of written records on how a program was constructed, what it does, how it does it, how to use it, and how it interacts with other programs[30]. Software documentation should be considered an integral part of software design and not an add-on component; however, this is rarely the case.

Software documentation is a very broad area. This chapter briefly discusses all the areas but proceeds to place particular emphasis on documentation that assists a programmer in understanding a program at the code level.

## 2.1 How is Software Documentation Perceived?

Software documentation is one of the least glamorous and least favoured activities in software engineering. It is often an activity that is postponed by programmers until the last opportunity or indefinitely. This attitude may come about because the success of a development is often judged by performance and cost criteria alone. Due to a lack of foresight, little credit is given to a program that will be easy to maintain in the future. This attitude is less prevalent today than it was perhaps ten years ago as the reality of maintaining poorly documented and increasingly complex systems has enlightened many companies through experience. Unfortunately, this is not a universal situation, as development and maintenance is often performed by different teams with little communication between them.

When it becomes obvious to management that a project is running behind time during development, corrective actions are often taken. The futility of recruiting new staff at a late stage in a project, is well established[12]. The alternative is to eliminate all those activities deemed nonessential to the finished product. Invariably, software documentation is the first to suffer. Often the intention will have been to defer the documentation until after the development, but in practice this means it will never be completed.

The quality of the documentation supplied with a product can only be ensured when the management and the customer appreciate the long term benefits to be gained from it.

I believe the poor image of documentation amongst the software engineering community is mainly due to programmers experiences with existing documentation and the unremitting burden of maintaining consistency between the source code and the documentation.

Existing documentation, especially that found in the maintenance phase of the life-cycle, is often out of date, incomplete and difficult to access. The unreliability of documentation can cause inexperienced programmers to be misled while those more experienced view the documentation with the scepticism it deserves.

The highly adaptive nature of software, compared with other engineering and scientific disciplines, means that a programs documentation requires continual maintenance in order to keep the two in phase. If the specific problems of software documentation are to be

overcome, then specialist tools are required. To date, most software documentation has been produced using general purpose word processing and graphical packages that offer no assistance to the problems of software documentation. The recent advances in documentation preparation systems has been in presentational issues — graphics, colour, fonts, laser printing, page layout and the like. Little progress has been made at the level of structure, update and retrieval. Todays latest desktop publishing systems are only doing, albeit in a glossier wrapper, what simple word processing systems were doing 10 to 15 years ago — communicating information on paper.

## 2.2 Important Qualities for Software Documentation

For software documentation to be useful it should aim to possess a number of qualities. These qualities, as discussed in this section, may appear common sense, but it is difficult to find documentation that satisfies just a few of them. Most are general in nature and could be applied to technical documentation from any discipline.

### 2.2.1 Readability

Documentation should be clear, precise and easy to read. Otherwise it performs no purpose as the source code itself may be as readable as the documentation. Any notations or formal languages used in the documentation should be adequately explained if they are not described in other literature.

### 2.2.2 Maintainability

Due to the flexibility of software, a program undergoes many changes during its useful life. Many of these changes are the result of adaptive maintenance which alter the functionality of the program or allows the program to run in a modified environment. These types of changes, more so than other types, will require the documentation to be updated to maintain consistency between the program and its documentation. If documentation prepared during

the design of a program is to be of use during software maintenance its benefits must outweigh the cost of maintaining it alongside the code.

The burden of maintaining the documentation in parallel with the code is avoided with products that produce documentation automatically from static analysis of the source code (see Section 3.2).

### 2.2.3 Suitability

Documentation needs to be tailored to its audience: there will be a wide range of staff with differing levels of experience and ability; documentation must be capable of satisfying the needs of all these people.

Information hiding is an approach used in software system building[51]. Each module of a system designed in this way hides the internal details of its processing from other modules that use it. The same term, information hiding, can be applied to documentation that is organised to allow it to be read at differing levels of detail. High level documentation should avoid discussion of low level detail. For instance, an overview description in a document would hide low-level and detailed descriptions from the reader. But, the capability to access low-level detail should be provided when required. In conventional technical documentation this is normally achieved by cross-references of the form: *see page nnn.* Cross-referencing allows common information that may be used throughout a document to be centrally located. This feature simplifies the update of information in future versions of a document.

The power of cross-referencing is widely acknowledged. Hecht[52] discusses the need for forward and backward referencing between levels of documentation; the high-level documents point to the lower level documents and vice versa. He also suggested that cross-referencing may help identify the areas of the documentation affected by a change to the software and that automation may help in the referencing effort. James[42] highlights the importance of cross-referencing in technical publications between related information.

Software documentation should place greater emphasis on why and how something was

done rather than what has been done[38] since the source code itself accurately describes what has been done.

### 2.2.4 Redundancy

If a piece of information can be derived from another source then it is redundant. In software documentation redundancy may occur because the information is duplicated elsewhere in a document or the information can be generated automatically from the source code.

An example of redundant documentation can often be seen in assembler source code where each assembler instruction is commented with a description of what the instruction does: an assembler mnemonic such as *'INC A'* may have the comment *'increment the accumulator'* associated with it, which conveys no more information than the mnemonic itself. Singleton[56] quotes a study of the documentation of a large program that showed that there was 70 percent redundancy in its software documentation.

### 2.2.5 Consistency

Consistency is the quality that ensures that all abstract representations of a program: source code, design, specification, etc., do not contradict each other. Sommerville et al.[58] states that we are unable to use a software tool to automatically check that all representations of software components are logically consistent since this is beyond the capabilities of the science. In the system they developed, consistency checking is limited to checking that if one representation is modified, all other associated representations are also modified.

### 2.2.6 Completeness

Completeness is the property which describes the coverage of the documentation over the software that it describes.

In software maintenance it is rarely necessary to have 100% completeness for the documen-

tation since many parts of a program are never examined or modified. The 80/20 rule often applies: 80% of the time is spent on 20% of the code. The requirements for a redocumentation system given in this thesis (Section 4.3) specify that it should support the production of incremental documentation, which by its definition allows less than 100% completeness.

### 2.2.7 Traceability

Traceability in program documentation is the property that links related components from different phases of the software life-cycle. This property, for instance, makes it possible to trace from the requirement of a particular function in a system, through its specification, design and then to the actual program code that implements that function. The links need not necessarily stop at the program code, as traceability can also be provided to the testing scripts for the function and to defect reports related to the function. Schneidewind[54] considers this an important property of documentation and Mullin[49] states "traceability is the key to Product Assurance".

### 2.2.8 Accessibility

James[42], in talking about the communication of technical information says.

> A vital article buried in a stack of irrelevant paper is almost as unavailable as if it had never been written.

The size of the retrieval problem is proportional to the quantity of documentation to be accessed. The usual techniques for retrieving information from documentation are: use of table of contents and index, skim reading, full text reading and text string searching (on-line documentation only). Skim reading and full text reading are only practical for locating information when the document set is small. Also skim reading is less effective for on-line documentation as the process of flicking through the pages of a book is difficult to model on a VDU screen. Retrieval by text string searching enables a reader to locate all the points in the documentation where words or phrases of interest occur. To date, this method of

retrieval is the only widely used one to make use of having the documentation on-line.

## 2.3 Economics of Documentation

Software documentation has already been identified as a major contributor to the high costs of design[10] and maintenance[46]. According to the COCOMO database of several hundred software development projects[10] the following statistics were calculated for documentation effort in a project.

- It takes on average three hours to produce a page of software documentation.
- A rough estimate of documentation effort showed that one man-month is spent on documentation per thousand source code instructions.
- About 51–54% of a software projects effort results in documentation as its immediate end product.

These figures can only be taken as a rough guide due to the difficulty of obtaining consistent and reliable data. But, they emphasise the potential financial gains that can be made by applying the use of efficiency improving tools to the documentation effort.

## 2.4 How Much Documentation?

The amount of documentation required for a project is difficult to estimate because of the number of variables involved. It is influenced for example by: the size, complexity and structuredness of the source code; the use of high or low level languages; and multi or single user. These factors are all related to the source code but a major influence on quantity is the requirements of the customer and the management of the development organisation. These issues can be summarized as follows:

- Formality, extent and level of detail required.

- Responsibilities and schedules for documentation preparation.
- Procedures and schedules for documentation.
- Review, approval, and distribution.
- Responsibilities for documentation maintenance and change control.
- Audience for which the documentation is intended.
- Amount of redundant information in the documentation.
- Rigidity of the document guidelines and standards.
- Number of users effected by the documentation.
- Frequency of use of the software.

It is usual to find that a software project has incomplete documentation, however, it is possible to over document software, especially when it involves redundant information[4, 13]. The repeated presentation of the same information may obscure other unique and more important information[13].

A balance must be achieved between keeping the amount of documentation to a minimum, because of its high cost, and describing the software in sufficient depth for the intended audience. It must be remembered that while preparing documentation is time consuming and expensive; reading is more expensive because more people are usually involved.

### 2.4.1 Reducing the need for Documentation

A number of approaches can be used during program design to reduce the need for documentation. They are mainly effective at reducing the need for low level program documentation.

- Good design methods and practices.
- Structured programming.
- High-level control structures.

- Meaningful identifier names.
- Selective high level code commentry.
- Consistent style.
- Self documenting/High level languages.

Martin[46] suggests the ultimate solution to the documentation problem is self documenting programs. No current programming languages meet this criteria, although many, especially 4GLs, have made this claim.

## 2.5 Documentation Activities

The main activities required for documentation are.

**Creation** Creating the original documentation during development and creating 'retrospective' documentation[30] during maintenance. The later activity has been named 'redocumentation' in this thesis. It includes creating documentation where it is nonexistent and replacing documentation which is so out of date that it is not worthwhile updating.

**Update** Updating the documentation to reflect the results of the improvement effort during design and maintenance. All documentation should be kept up-to-date during all phases of the life-cycle. This may not always be possible during maintenance in situations where the original documentation is too inaccurate or inaccessible for cost-effective maintenance. An exception to this rule can be made for documentation prepared during the initiation phase of a project since it is a short term document containing an analysis of the project when it was initially proposed and has no long term significance.

In many cases more problems are encountered changing the documentation than changing the software. Changes to the software are therefore often quicker and cheaper to implement than changes to the documentation.

## 2.6 Main Categories of Software Documentation

### 2.6.1 Internal Documentation

Internal documentation is embedded in the program using the commenting facilities of the programming language. It is common practice to have project standards for header comments that explain the purpose of modules and routines.

Many embedded comments are found to be inconsistent with the code and can therefore cause great confusion to a maintenance programmer. Which is correct?: the comments or the code. Inconsistency results when changes are made to the code without equivalent changes being made to the comments. Martin and McClure[46] said:

> If a program is well structured and properly documented internally, the program source code can provide all the necessary program documentation.

This statement is difficult to agree with since there are many types of documentation that cannot easily be conveyed using comments embedded in the source code. For example, the source code does not seem the most appropriate place for organisational and overview documentation.

### 2.6.2 External Documentation

External documentation is separate from the program and has two subcategories; independently and automatically generated.

#### Independently Generated

Most development documentation falls into this category. It is produced manually ,usually with the assistance of word processsing and graphical packages. Often it is discarded once development is complete because it is considered unnecessary and too expensive to

update. Maintenance programmers tend to distrust it because they know that it is rarely updated[46]. Sneed[57] reports on a software system where the external documentation was scrapped because it was so incomplete and out of date that it was not worthwhile using during maintenance of the system.

### Automatically Generated

The most accurate software documentation is that generated automatically from static analysis of the code. It requires no maintenance effort itself, apart from the machine time to reanalyse the code, since it is updated overnight following a change to the code. Tools that generate documentation of this type produce for example: cross-reference listings, module hierarchy charts, control flow graphs and calling hierarchy charts.

Different types of information is being extracted from the code and presented in a more compact and comprehensible form than in the code itself. Many of the commercial tools in this category claim that they satisfy all the documentation needs of software maintenance. Although the information they produce is of significant use to the maintenance programmer, they do not provide information about the design of the system. This is only available in the original documentation, if any exists, or in the heads of the designers who developed the system, if they can still remember.

## 2.7 Software Documentation Standards and Guidelines

There have been a number of documentation standards and quidelines produced[1, 2, 5, 6, 27], but unfortunately none seem broad enough to be applicable to all situations and they often have little relevance to the application area being considered. They should however provide some basis from which project specific documentation standards can be generated.

Many documentation standards appear out of date when compared with the progress that has been made in other areas of computing over the last ten years. A recent documentation guideline from ANSI, titled 'Guidelines for the Documentation of Digital Computer Programs'[5] illustrates the inadequacy of such literature. The advice given in the guideline

is very general in nature and the complete document is only three pages in length. If a standard or guideline is to define what information should be expressed and how that information is to be presented in software documentation, then three page cannot be enough.

## 2.8 Why is Software Documentation Important?

Without software documentation, programmers must rely on the source code to provide all the information they need to maintain a program. Unfortunately current programming languages do not embody all this information as they record no knowledge about why particular design decisions were taken. Therefore software documentation must be provided to communicate this knowledge.

Like programs, documentation must be considered an important product. Documentation is as much part of a product as the hardware and software[42].

## 2.9 The Software Project Document Set

The following subsections give an overview of the types of documentation that should be produced for a software project during its life-cycle.

### 2.9.1 Documentation Prepared During the Initiation Phase

Prior to starting a software project it is normal to perform a study to access the value of the project. The following documents would be produced during this phase:

#### Project Request Document

Provides the means for a user organisation to request the development, procurement or modification of software or other related services. It is the initiation document of the

project life-cycle.

#### Feasibility Study

Provides:

- Analysis of objectives, requirements and system concepts.
- Evaluation of alternative approaches for reasonably achieving the objectives.
- Identification of proposed approach.
- Preliminary user documentation.

A Feasibility study in conjunction with a cost/benefit analysis should provide sufficient information to allow management to make decisions on the future of a project.

#### Cost/Benefit Analysis

Provides adequate cost and benefit information to analyse and evaluate alternative approaches and make decisions to initiate or continue the project.

### 2.9.2 Documents Prepared During the Software Life-Cycle

Each documentation type is a by-product of a phase in the software life-cycle.

#### Operations Documentation

Provides computer operation personnel with a description of the software and the operational environment so that the software can run.

### User Documentation

Describes the functions performed by the software in a terminology appropriate to the expertise of the user. The quality of this documentation has a significant affect on the usability of the software.

Good user documentation may help resolve questions about what the system should or should not do in the absence of a specification. This is important when trying to determine what category of maintenance activity a user request for change falls within.

### Program Documentation

The different levels of documentation give different views of the program. A wide range of graphical and textual methods are available for presenting program documentation.

- **Requirements Documentation**

  Forms the basis of the mutual understanding between the users and the designers of the functionality of the software. Includes the operating environments and development plans.

  Provides data description and technical information about data collection requirements.

- **Specification Documentation**

  Specifies for the analysts and programmers the requirements, operating environment, design characteristics, and program specifications for a system or subsystem.

  Specifies for the programmers the requirements, operating environment, and design characteristics of a computer program.

  Specifies the identification, logical characteristics, and physical characteristics of a systems database.

- **Design Documentation**

  The documentation produced as a by-product of the particular design methods and strategies used during development.

- **Implementation Documentation**

  The boundary between design and implementation documentation is indistinct and the two types of documentation tend to merge together. There are three levels of implementation documentation: program overview, program organisation and program instruction. Each of these levels and their components are discussed below.

  **Program Overview** Overview documentation provides an introduction to the program. It is often of use in providing new maintenance staff with the basic knowledge they need to start maintenance work on a program. It describes the program in a broad, abstract way, and tends to be the most stable type of implementation documentation: most post development changes do not change the central structure of the program. In large programs it is used by experienced maintenance staff working on localized areas of the program who need information about other parts of the program that they are unfamiliar with.

  Martin and McClure[46] claim that overview documentation is brief and simple to produce. It is certainly simple to produce if it is created during the early design stages of a program; unfortunately this is not usually the case. Maintenance programmers then have to attempt to abstract the overview from the source code. Any one who has tried this for all but the smallest of programs, will appreciate the difficulty of this process.

  Where overview documentation has been produced at early stage during the design of a program, it is often used as a discussion document for the determining the design of the program. In many cases it is never updated to reflect the actual design used in the program. The maintenance team is then presented with an overview document that only represents a proposal for the program and not the actual program developed. The frequency with which this situation occurs may indicate the degree of difficulty associated with updating overview documentation.

  **Program Organisation** Organisation documentation describes the structure of the source code and its interactions with its environment. It will contain information about: module hierarchies, inter-module relationships, module level commentry, data structure commentry and; hardware and operating system interactions.

**Program Instruction** Program Instruction documentation is the lowest level of software documentation. It describes what named items in the source code are used for and, how and why they operate. It is particularly important to provide internal documentation for 'clever' areas of code and for code where the operation is unclear.

Letovsky and Soloway[45] proposed the 'role' and 'goal' approach to documenting variables. The role describes what the variable is used to hold in the program and the goal describes what the variable achieves in the program.

Implementation documentation typically includes source code commentry, data dictionaries, flow charts, state transition diagrams, etc.

- **Testing Documentation**

  Provides a plan for testing the software; detailed specifications, descriptions, and procedures for all tests; and test data reduction and evaluation criteria.

  It should describe the test analysis results and findings, present the demonstrated capabilities and deficiencies for review, and provides a basis for preparing a statement of software readiness for implementation[6].

- **Maintenance Documentation**

  Provides the maintenance programmer with the information necessary to understand the programs, their operating environment, and their maintenance procedures. A separate document is not always necessary here as this information should be available in the other documents.

### Historic Documentation

Historic documentation records the evolutionary path of a program throughout its life and ensures important design and maintenance information is available to the current maintenance team. It will typically consist of two documents; a system development journal and a system maintenance journal[46]. The content of these would be:

- **System Development Journal**
  - Development philosophy.

- Decision making strategies used.
- Reason for a particular design.
- Project goals.
- Priorities.
- Experimental techniques.
- Tools and how they were used.

- **System Maintenance Journal**
  - Change philosophy.
  - Quality preservation/improvement strategies.
  - Problems.
  - Trouble spots.
  - Change/Error history.

# Chapter 3

# Survey of Software Documentation Tools

## Introduction

This chapter surveys commercial and research tools that are specifically oriented towards the production and support of software documentation. Examples of desktop publishing systems and other documentation technologies have been examined as part of this research, however they appear to offer no significant improvements over their predecessors for software documentation and are therefore not covered in this survey.

Until recently there has been very little research into software documentation tools. In Europe, both the Esprit and Alvey programmes have only supported a few projects where software documentation is a major issue.

Of the tools available commercially, most fall within the category of automatic documentation and not in the area of creating and managing manually created textual and graphical documentation.

Even the latest generation of IPSEs and APSEs treat the production of software documen-

tation in a simplistic way. They provide a collection of general purpose tools for document production and modification, and then provide an interface to the environments database for configuration control. They provide limited technological support for achieving the qualities described in Section 2.2.

## 3.1 Software Documentation Environments

Over the past few years a number of software documentation environments have been discussed in the literature[34, 40, 47, 58] that support the production, management and use of textual and graphical documentation during all phases of the software life-cycle. Most of these documentation environments provide facilities to support traceability, central storage of all the projects documentation, easy access and update, and the enforcement of project wide standards on the structure of the documentation.

These documentation environments provide useful facilities for the production of conventional life-cycle documentation during the development of a project, but they are of little use to programmers faced with a completed system that has little or no existing documentation. Presented with such a problem during maintenance, it is not usually considered economically feasible to reproduce the development documentation from scratch, which is the approach that would be needed if one of these environments were to be used.

The following sections discuss a selection of these environments and Table 3.1 compares their features.

### 3.1.1 FORTUNE

FORTUNE is a collaborative project forming part of the Alvey Software Engineering program. Its aim is to produce an integrated documentation tool that will support the creation and update of textual and graphical documentation throughout a projects life-cycle.

FORTUNE is based on the traditional life cycle model. Conventional development documentation is created in the system during each phase of the life cycle. This may include

specification documents, data flow diagrams, data dictionaries, and program source code according to the development methods in use. The ability is provided to support links between different levels of documentation, known as traceability(Section 2.2.7). For instance, it would be possible to relate a component within a requirements document to a related component within a specification document which would in turn have a link to the related design documentation and so on. As another example, a section within a maintenance change document may have links via the testing, code and design documentation back to the area of the specification relevant to the maintenance change.

The following list includes the issues consider important by the FORTUNE consortium from available documentation[47, 49, 48]:

- It incorporates a structured graphical editor that can be configured to support a range of graphical design methodologies. However, it does not perform consistency checks on the design as available on some PC based products that support design methodologies. It is proposed that further tools can be supplied to perform this checking on the design via the Public Tools Interface.
- Allows documents to be created and edited.
- Generic document structures can be defined at the beginning of a project by management. All documentation must then conform to these structures.
- It will support traceability between levels of documentation as discussed above.
- A Public Tool Interface to FORTUNE will be provided to allow stand-alone tools to operate upon FORTUNE documents.
- FORTUNE will initially be sold as a stand alone tool but may later be integrated into other manufacturers IPSEs.
- A textual interface to FORTUNE will be provided to allow it to be run non-interactively.
- FORTUNE will be integrated with a configuration management system.
- FORTUNE will support the production of text associated with mathematical based methodologies(e.g. Z and VDM). Any manipulation of the mathematical expressions will be performed via the Public Tool Interface.

FORTUNE has the potential of being a useful documentation tool for designing new systems. It enforces standards on the documentation and allows all the development documentation to be centrally located, with easy access and update provided. Documentation provided in this form is likely to be a major factor in reducing the cost of software maintenance.

FORTUNE is of limited use for the retrospective documentation of existing systems during software maintenance as it does not have the capability of incremental (Section 4.3.1) update of the documentation database in an informal manner. It would be necessary to redocument the whole system before any gains could be achieved in the maintenance phase. As mentioned before, this is usually prohibitively expensive

### 3.1.2 SODOS

SODOS is a software documentation support environment that was developed as part of a Ph.D. dissertation at the University of Southern California in 1984[40, 41]. It is based on the same philosophy as FORTUNE and DIF, but lacks the graphical support of FORTUNE. According to papers published[40, 41], SODOS has been implemented in Smalltalk-80.

### 3.1.3 DIF

DIF[34](Document Integration Facility) departs slightly from the other environments in that it has the additional aim of integrating documents within and across several projects into a single environment. It was designed for use within an experimental System Factory developed at the University of Southern California to study the development, use, and maintenance of software systems. Like FORTUNE and SODOS, this environment is designed to support the production of documentation associated with the phases of the traditional software life-cycle model. The particular model used here has eight phases and includes a maintenance phase.

Software documents are decomposed into segments which can be considered as objects to be stored, processed, browsed, revised and reused. Links are used (hence it is considered

a hypertext system) to define the relationships between objects. Each object is stored within a separate file in the UNIX filing system. The file system is used for the hierarchical relationships between objects and a relational database is used for the nonhierarchical relationships. The authors claim that:

> ...judicious use of links alleviate the problems of traceability, consistency and completeness.

This may be true, but it exemplifies one of the fundamental problems with authoring hypertext systems and that is that the quality of the documentation is dependent of the skills of the original author in creating links at logical points in the documentation. Two hypertext documents that contain exactly the same textual content, but organised by different people may appear different in terms of both quality and lucidity to an end user. At the current stage of hypertext research there are no automated strategies for ensuring that links are created in the correct quantity and at the correct position in hypertext documents.

### 3.1.4 SOFTLIB

SOFTLIB[58] is a documentation library system based around the UNIX file store. It was developed with the aims of demonstrating ideas on the management of software documentation associated with large software projects and to access the usefulness of limited forms of completeness and consistency checking. It is a stand alone library system and does not provide tools for document preparation

Sommerville et al. argue the case for documentation based on software components rather than the software life-cycle approach of DIF, FORTUNE and SODOS. A software component is any software item that has an associated specification. In this way all documentation for a component is grouped into a set. Making it easy to trace the different representations of a component. The authors claim that this approach encourages software reuse and makes limited completeness and consistency checking easier.

SOFTLIB does not provide any revision control mechanism, although a transaction log is maintained by the system. The term 'version' is used to describe components with a common

abstract specification, but whose implementation-dependent representations are different. For example, a specification for a stack abstract data type may have implementations in C and Pascal.

The approaches taken for completeness and consistency checking are quite simple. Each software component has predefined set of required documents. The document library achieves completeness checking by ensuring that when a document set is placed in the library following its creation or update, all the required members of the set are present. As full consistency checking is not possible(See section 2.2.5), the approach taken in SOFTLIB is to check for inconsistency between representations. This is achieved by insisting that when one representation of a software component is changed, that all other dependent representations must also be changed in the same editing session. SOFTLIB does allows short term inconsistencies for the 'quick fix'.

### 3.1.5 Symbolics Concordia and Document Examiner

Symbolics supply their software product documentation, which amounts to the equivalent of 8 000 pages, in a hypertext format. Unlike many hypertext systems they have chosen to separate the tasks of writing and reading the documentation by providing distinct tools.

Concordia[62] is a documentation development environment that allows technical writers to create a hypertext database of documentation. This database is then viewed by the user via a delivery interface known as Document Examiner[63].

Users can navigate around the database by moving from node to node in the hypertext. An overview command allows the users to see the context of the current node in relation to its parents, siblings and children. This assists the user in determining if the information in the current node is relevant.

Comprehensive string searching is provided for locating information. It includes heuristic matching against title and keywords of nodes in addition to exact and substring matching.

This documentation system is primarily aimed at producing and viewing technical documentation and has been successfully used by Symbolics to supply the manual set on their

| Property | Documentation Environment | | | |
|---|---|---|---|---|
| | DIF | FORTUNE | SODOS | SOFTLIB |
| Tools Interface | yes | yes | no | no |
| Revision Control | external | external | internal | none |
| Graphical Support | external tools | yes | external tools | external tools |
| Generic Document Structures | yes | yes | yes | no |
| User Interface | ? | mouse and batch | mouse | menus |
| Document Retrieval | DBMS query language | menu | menu | menu |
| Traceability | yes | yes | yes | yes |
| Completeness | yes | yes | yes | yes |
| Consistency | yes | yes | yes | yes |
| Access Control | ? | ? | ? | yes |
| Document Organisation | life-cycle | life-cycle | life-cycle | software component |

Table 3.1: Comparison of software documentation environment features.

Lisp machine for several years. It is a general system and could be used for any technical/user documentation. However, its separation of the reader and writer role makes it mainly suitable for documentation where the period between releasing updates of the documentation is in the order of months or years rather than days or weeks. Thus making it unsuitable for source code documentation.

## 3.2 Automatic Documentation Tools

There are many tools available, especially in the COBOL programming domain, that fit into the category of automatic software documentation[7, 22, 39, 43, 57]. These tools perform a static analysis of the source code to produce a series of reports. These reports include cross-reference listings, metric reports, and module hierarchy charts. The information they provide helps the programmer in understanding the structure of the system and how components within that system interact. Since the documentation is produced directly from the source code the only effort required to keep the documentation in step with the evolving source code is to rerun the tool over the new version of the system. This operation would normally be performed in batch mode, overnight. Several of the vendors of these tools lead us to believe that documentation generated in such a way is the total solution to the doc-

umentation problem. However, our experiences have shown that although this information plays an important role in improving the efficiency of programmers involved in maintenance work, it does not directly assist in the comprehension process.

Examples of typical tools of this class are discussed in the following subsections.

### 3.2.1 SOFTDOC: Software Documentation System

SOFTDOC is a typical example of a commercial automatic documentation tool. Sneed[57] discussed a software re-engineering project that used the system and Jandrasics[43] has discussed the system itself.

SOFTDOC can be used to analyse programs written in PL/1, COBOL or assembler. It generates listings which include the following information:

- Module tree diagram.
- HIPO diagram.
- List of external/internal interface.
- Control flow graph.
- Data reference table.
- List of test paths.
- List of symbolic constants.

### 3.2.2 The C Information Abstractor

The C information abstractor[22] collects information about C programs by static analysis of the code like other automatic documentation tools but instead of providing listings as output, the system stores the information in a relational database. High-level commands are provided to access the information in the database. Typical questions that can be asked by the user are:

1. Which functions call the function *xyz*?

2. Where is the structure *abc* defined?

3. Which functions use the global variable *lmn*?

4. What is the global constant *rst*?

The commands allow software objects to be displayed and examined. The database does not contain copies of the objects, but keeps a record of the module (compilation unit) containing the object and the range of lines in that module that the object spans. This is then used by the system to retrieve objects when a user requests their display.

The authors suggest that the system could be extended to allow structured comments to be used to record information that cannot be derived automatically from the code.

The system appears to be a cross-referencing system using a database to store information and provide an improved user-interface. The queries that the system answers are similar to those that can be answered using the prototype browsing system discussed in Chapter 6 but it does not provide as effective user-interface.

## 3.3 Other Software Documentation Tools

### 3.3.1 DOCMAN: Documentation Based on Cross-Referencing

A documentation system known as DOCMAN[33] has been proposed based on cross-referencing that aims to meet the needs of programmers maintaining large software systems. It allows documentation produced by maintenance programmers during the examination of source code to be linked with cross-referencing information obtained by parsing the source code. Three categories of documentation are catered for by this system:

**Encyclopaedia** This is the lowest level of documentation provided in the system. It consists of descriptive comments about the use of and/or operation of identifiers within

the source code. Whether they are routines, data structures, data items or constants, etc.

**Glossary** Within the documentation of a system there will be words and phrases with special meaning that appear frequently. The glossary documentation category provides the mechanism for documenting these words or phrases.

**Overview** This category of documentation provides the high level narrative that describes the system as a whole. This category is essential for the person new to a program who will find the low level information given by the encyclopaedia and glossary entries too detailed for the early stages of understanding a system.

The text entries for all three categories may refer to other encyclopaedia or glossary entries. Therefore, by means of scanning the documentation, each encyclopaedia and glossary entry can have generated a list of references to other parts of the documentation where that entry is referred to. A more detailed description of DOCMAN can be found in [33].

DOCMAN makes extensive use of cross-referencing between documentation entities, cross-reference listings of the source code and the source code itself. The system has been developed as both a hard-copy and an interactive system. The interactive component of DOCMAN allows the user to display information about selected names and to add to the documentation base, but it does not provide machine support for the traversal of the mass of cross-references that are created when dealing with a large program. DOCMAN shares some of the cross-referencing concepts of the general hypertext technology, but it does not offer as powerful interactive facilities. It does however, suggest useful concepts for the documentation of source code that could form the basis of a hypertext documentation tool that meets the requirements outlined in Section 4.3.

### 3.3.2 The Neptune Hypertext System

Hypertext systems have been surveyed in the excellent articles by Conklin[23, 24] and there is no need to repeat the information here. It would, however, be useful to have a look at one particular system that has a number of features that make it particularly suitable for the application considered in this thesis. The system is the Tektronix Neptune

system[25, 26, 19, 20] developed as a research project, but since sold to Mentor Graphics who may turn it into a commercial product.

The designers have split the Neptune system into two components: an application layer and a hypertext transaction server. The transaction server is called the Hypertext Abstract Machine (HAM) and is written in the programming language C. It provides a number of facilities useful for building large hypertext systems.

- Distributed access.
- Multi-user access.
- Transaction based crash recovery.
- The destination of a hypertext link can be an offset within a node.
- Link attachment may refer to a particular version of a node or it may always refer to the current version.
- Maintains a version history of each node and provides rapid access to any version.

Tektronix provided a graphical user-interface using the language Smalltalk-80 that communicates with the HAM using a set of defined operations. Application specific interfaces can be built in any language, communicating with the HAM in the same way.

The advantage of using a system such as the HAM is that applications can be built using hypertext principles without having to reinvent a database for storing hypertext structures.

### 3.3.3 The Smalltalk-80 Browser

The Smalltalk-80 browser commands; explain, comment[35, 44], inst var refs and class var refs[44] are analogous to facilities being proposed in this thesis for the hypertext redocumentation tool.

The explain command provides information about any single token within a method. The user selects the token of interest in the current view. Then via a menu option the command

explain is chosen and the system then inserts brief information about the token immediately after the token. Smalltalk-80 is the only programming language that allows embedded comments to be associated with the software object to which they apply.

Information can also be obtained about instance variables by selecting the variable of interest and choosing the menu option comment which, like the explain command, displays information about the selected item. Two other commands that provide information on variables are *ins var refs* and *class var refs* which display all the particular places an instance variable or a class variable are used. The system creates a new browser window with a list of the methods in which the variables occurs. By clicking on any of the names in the list, the corresponding method can be examined or updated.

# Chapter 4

# Redocumentation

## Introduction

Software documentation should be produced as a by-product of the development process and handed over as a complete package along with the source code to the team that will maintain the program. However, this is rarely the situation in practice. Most software reaches completion without useful documentation for the people that have to maintain it.

Redocumentation is the activity that many maintenance teams are forced into because the software documentation that is supplied with the program they have to maintain, is inadequate or nonexistent. It involves creating documentation by analysis of the source code by experienced programmers and the recovery of useful documentation from the original documentation.

## 4.1 Why do we need to Redocument Software Systems?

There are often a number of problems with the software documentation that is supplied to maintenance teams from the development phase of a software system.

The documentation for a large program will often consist of many filing cabinets brimming with paper documents. In a well administrated project these documents may well be organised and structured with a comprehensive indexing scheme. Unfortunately, this is rarely found to be the case. The maintenance programmer then has the unenviable task of searching through a mass of documentation for the information relevant to the area of their work. Problems are also encountered when trying to update such documentation.

Due to the absence of useful standards and guidelines on documenting programs, there are a wide range of documentation techniques in use. Here, as with the choice of document preparation system, the choice of documentation technique is in many cases arbitrary. Therefore the software maintenance programmer may have difficulty understanding the documentation if the technique used by the original programmer is unfamiliar.

Within a software maintenance team there is invariably a wide range of programmer experience and ability. Therefore the documentation should be capable of providing understandable information for all these levels of experience and ability. This can be achieved by providing documentation that spans from a broad overview of the program, through to the nuts and bolts of the implementation. Unfortunately most existing documentation does not meet this criteria.

Inevitably, as the design phase of a software project proceeds, especially towards its end, the pressures begin to increase on the staff to meet project deadlines. The result of this pressure is that activities considered nonessential for the release of the product are often postponed or they are dismissed as unnecessary. In many cases documentation is classified as such an activity. This attitude leads to the development documentation becoming incomplete and out-of-step with the software. In fact the scenario described is probably excessively optimistic considering that many programs being maintained today were developed before the current concern with software maintenance. These programs were often completed with virtually no useful documentation for software maintenance.

Today, there are many document preparation systems available for the production of software documentation. Even in a single department of an organisation there may be many alternative systems in use at any one time. Often the system used is based on the programmers personal preference or their familiarity with a particular system. Also, in recent

years there has been a rapid evolution of document preparation systems and their associated storage media. Combine these two facts and the result is that, in general, software documentation is frequently found to be written on a wide range of document preparation systems running on different hardware that may no longer be available to maintenance teams. Therefore at some stage in the programs life it becomes impractical and uneconomic to keep the existing documentation up-to-date.

Before expending effort on redocumenting a program it is important to assess the value of the original documentation to determine if it is worthwhile maintaining and what should be kept or thrown away. This can be achieved by examining the documentation for the qualities outlined in Section 2.2. The final judgement on the usefulness of the documentation will be an opinion based on experience as there are no metrics for assessing the value of documentation.

No research has been performed to determine whether the documentation produced during development is the best form of documentation for software maintenance. Unfortunately, like many areas of computer science, the experiments required to determine this would be prohibitively expensive. But a survey of maintenance programmers could establish if design documentation is used when it is both available and of good quality. From speaking to people involved in software maintenance, it would appear that overview documentation is often considered the most valuable form of design documentation for software maintenance. Other, more detailed documentation is not so popular. This may be because there is insufficient technology for maintaining the consistency of this type of documentation or because it is not appropriate to the cognitive processes involved in analysing code.

## 4.2 Current Approaches to Software Redocumentation

At present the number of choices available to a programming team faced with redocumenting a large program is limited.

One approach is to reproduce the design documentation from the source code. To be effective, the complete program must be documented or at least complete subsystems. This

requires a large number of a maintenance team to be tied to the redocumentation effort and therefore unavailable for the main stream activity of satisfying user-requests. The results of an experiment which took such an approach reported that it was expensive[57]. When dealing with a program having a predicted maintenance life of, for example, 20 years, this approach may be economically viable, but for programs with a short maintenance life it is unlikely to be appropriate. However, if the approach is taken, it would be advantageous to use a documentation support environment that managed the life-cycle documentation set to avoid repeating the problems outlined in the previous section. These environments are primarily aimed at those designing new programs, but they would also be of use in this approach to redocumentation. They enforce standards on the documentation and allow all the development documentation to be centrally located with easy access and update provided (See Section 3.1).

There are a number of tools available that claim to satisfy the documentation needs of software maintenance. These tools generate automatic documentation in the form of reports by static analysis of the source code. Examples of the documents produced are: control/data flow charts, cross-reference listings, metric reports, call graphs and module hierarchy charts. All this information is of significant use to the maintenance programmer in becoming familiar with the structure of a program and in navigating around the program during maintenance investigations. What they fail to do is provide any insight into why particular program structures are used or why certain design routes were taken. This knowledge can only be recovered by eliciting information from the original designers or by detailed examination of the source code by programmers. The advantages of these tools are that they are inexpensive to operate and the documentation produced is easily kept up-to-date. Unfortunately the majority of tools of this type are only available for analysing COBOL source code.

Code restructuring tools are worthy of mention in this section. Although they are not redocumentation tools in their own right, there use may be considered on the assumption that the structured code they produce will be easier to document than the original code. Given two programs with the same specification, but designed with different levels of structuredness, then the more structured design of the two will undoubtedly be easier to document. But you cannot necessarily extrapolate from this fact that code passed through a restructurer

will be easier to document than the original. A large part of analysing code during software maintenance work is trying to determine why the designer took a particular approach in the code. If code has been restructered prior to analysis then the original design will be obscured and therefore more difficult to determine and document. Restructuring tools should therefore be used with care during software maintenance.

## 4.3 Requirements for a Redocumentation System

At the Centre for Software Maintenance(CSM), University of Durham and British Telecom Research Laboratories(BTRL) the activity of redocumenting software systems has been recognised as an important area of software maintenance. From contacts made between the CSM and other organisations in the industry, it would appear that this view is wide spread.

The collective experiences of personnel involved in maintaining large software systems in a wide range of organisations has enabled the establishment of a number of key requirements for a redocumentation system for capturing the knowledge gained by maintenance programmers while analysing source code. These are discussed in the following subsections[31].

### 4.3.1 Incremental Documentation

The ability to build up the documentation for a system over a period of time in an incremental manner without the need to document the complete system in one step. This is possibly the most important requirement of any redocumentation system as it allows the documentation to be produced as code is examined during day to day software maintenance activities. The documentation can then become a byproduct of the maintenance process and not an activity in its own right.

Another benefit is that only the code that is analysed by maintenance programmers gets documented. No time is spent documenting code that is in a stable state and never examined or modified. It has often been said that the 80/20 rule applies to software maintenance; 80 per cent of the time is spent on 20 per cent of the code. Therefore it is unproductive to

document a complete system during software maintenance.

### 4.3.2 Informal Update

It must be easy for a programmer to add to the documentation as the source code is examined. The system should provide a 'notepad' like environment for creating documentation.

If the creation of documentation interferes with the comprehension process then the output of the maintenance team will be reduced and programmers will therefore avoid using it.

### 4.3.3 Quality Assurance

It is common practice within industry to perform quality assurance on changes made to the source code[29, 28] to reduce the possibility of introducing errors into the code. Likewise, the same procedures should be applied to the creation and update of documentation.

Quality assurance checks can either be made at fixed time intervals, when a certain number of changes have been made or prior to building a new version of the software. To achieve QA, all documentation created or updated needs a status attached to it with the name of the author of the change, a time stamp and a status of approved or unapproved.

### 4.3.4 Integrated Source Code

The system should integrate the source code with the documentation to allow the programmer to access the documentation while examining or modifying the source code. With conventional terminals it is very difficult to view both code and documentation in parallel. However, with the increasing popularity of large screen workstations it will become possible to provide a user interface that allows both source code and documenation to be viewed concurrently.

### 4.3.5 Integrated Automatic Documentation

There are a large number of tools on the market that produce automatic documentation from static analysis of the source code. These tools do not always meet the claims made for them in their advertising, but they do generate useful information about the source code in the form of reports. The information generated for a large program can be overwhelming. A redocumentation system should make use of this information and provide an improved user interface to the information.

### 4.3.6 Configuration Management

Configuration management(CM) must be supported to allow the documentation and source appropriate to a particular version of the system to be recovered[17] and for details of changes to be logged. This could be provided by the system itself, an underlying database management system with CM capabilities or, if the system were incorporated into an IPSE, its database could be used.

### 4.3.7 Team Use

A maintenance team working on a large program will have many members. For very large systems the number can be in the hundreds. Therefore a redocumentation system for use during software maintenance must support concurrent access and update by a team.

Multiple levels of access dependent on user status may be required. Some users may only require read access while others will need read and write access to the program and its documentation. Controlled access can also be used for sensitive programs like those in military equipment. Access to certain areas of the program and its documentation may only be allowed to those with the necessary security classification.

### 4.3.8 Information Hiding

The documentation for a large program will be immense. Information hiding allows the documentation to be read at different levels of abstraction from the implementation that it describes. It filters information. This is important when a maintenance team is composed of programmers with a wide range of experience and ability and particularly when new staff are becoming accustomed to a program.

# Chapter 5

# Hypertext and Software Redocumentation

## Introduction

This chapter describes the use of a commercial hypertext tool to demonstrate ideas for browsing and documenting programs. These ideas were originally aimed at providing software maintainers with a system which would allow them to document programs without interfering with their day to day functions of satisfying user requests for enhancements and defect removal. However, the same techniques could equally be applied to the production of original documentation during development.

The proposed system offers software maintainers an efficient and cost effective way of documenting a program where the existing documentation is inadequate.

## 5.1 Hypertext

Hypertext is a simple concept for the computer support of textual and and graphical documents. As an idea it has been around for many years[18], however, only recently has it received widespread attention following the release of a number of hypertext tools for microcomputers. Until then the concepts of hypertext had only been available on large machines for use in specialist applications. Hypertext supports links between related documents and allows users to browse the documents and traverse the links. A document can be considered as a set of nodes with links between those nodes to form a graph. Each node contains graphical or textual information. As an example, a hypertext user may be browsing a section of a document when a word or phrase is encountered in that document that is highlighted, known as a 'button'. This indicates to the user that a link exists to a related document. If the user then chooses to open that link, the display will be replaced with the document that the link points to. The new document will be related to original word or phrase in some way. It may be a more detailed description, a glossary entry, or perhaps a related subject area. The new document may also contain links to other documents. The actual details of how a hypertext document is browsed and the form of the links is dependent on the actual implementation. Conklin has published an extensive survey of hypertext systems[23, 24].

Although the underlying concept is simple, there is much research interest in how this concept can be used to provide solutions to problems in areas as diverse and complex as computer aided learning, public information systems, critiquing, authoring systems and computer based documentation.

## 5.2 Hypertext and Software Documentation

A number of tools have been developed that use hypertext principles to support the production of the document set associated with the software life-cycle during a projects development[34, 40, 41, 47]. These tools share a similar conceptual organisation of documents; they decompose documents into hierarchical structures of objects and use links to provide both the hierarchical structure of the document and traceability between objects in adjacent life-cycle phases. In this way, an object in a requirements document that describes

a particular feature of a system can be traced to its corresponding object in the specification document which in turn can be traced to the object in the design document that describes the features design. This linking can continue through all the phases of the life-cycle. The individual tools were discussed in more detail in Chapter 3.

From a survey of the literature on software documentation and hypertext systems, it would appear that no other researcher's have applied hypertext technology to the area of software redocumentation for maintenance. Yet the problems in this area are at least as large as those of producing documentation during software development. Hypertext has the potential of being a useful basis for the development of a system for the redocumentation of existing software systems. The power of cross-referencing between related components of documentation and between differing levels of documentation has already been recognised as valuable in hard-copy software documentation[33, 52]. Hypertext as a technology offers the capabilities of integrating these ideas into an interactive environment.

As mentioned above, systems have been proposed and developed that include some of these ideas for the support of software documentation production during the development of a project. Although valuable for new projects, this work has offered no solution to the problems of documenting software during maintenance. With the recent publicity[23] and availability of generalised hypertext systems it became obvious that the hypertext approach to supporting links between objects could be used to support the cross-references generated in the paper based redocumentation system, DOCMAN (See Section 3.3.1). The following sections discuss the experiences of using a commercial hypertext tool to identify how the machine supported links of hypertext can be used to enhance the interaction of DOCMAN with the programmer and to enhance its usefulness as a source code documentation system.

## 5.3 Scope of Research

The primary aim of this research is to demonstrate how the concepts of DOCMAN can benefit from incorporation into a hypertext framework. Although, at an early stage it appeared that hypertext had the potential for enhancing the interaction between the user and DOCMAN, it was still however necessary to create a tangible system for the further

development, evaluation and demonstration of these concepts.

The investigations have been performed by redocumenting a C cross-referencing program, XCC, developed at British Telecom Research Laboratories as part of the DOCMAN suite of tools. This is an 8 000 line program written in C. Although the size of the program is small when compared with the majority of programs being maintained in industry, it is believed to be of sufficient size for the investigation of approaches to redocumenting software systems. Nevertheless, at all times during this research the problems associated with mapping these ideas on to large systems with 200 000 plus lines of code, have been considered.

Two approaches to this research were available, either develop a prototype hypertext system of our own or use one of the generalised hypertext systems that has been developed commercially. A prototype system has the advantage that it can be adapted to meet specific needs as they arise whereas a generalised system is restricted to the facilities that the manufacturer has seen fit to provide. The proposed hypertext structure of the redocumentation system has links that can be created automatically. With our own hypertext tool it would be easy to write a program to create these links, but with a commercial system this would be difficult unless facilities are provided by the manufacturer to do so. The commercial system approach has the advantage that experience of hypertext technology is gained and it enables the establishment of requirements for a full-blown hypertext based redocumentation system without the commitment of producing code. After considering these factors, a two stage approach was chosen: firstly, use a commercial system as a mechanism for developing initial ideas on redocumentation and gaining experience with hypertext technology; then develop a prototype redocumentation system based on the knowledge gained from the first stage (Discussed in Chapter 6).

## 5.4 Choice of Commercial Hypertext Tool

The number of commercial generalised hypertext systems available at the moment is limited. The two most common systems for PCs are HyperCard[8] and Guide[14, 15, 50]. For the ideas being presented here it is necessary for the hypertext tool to support buttons embedded within the text of a document and to have links whose destination can be a region or a point


Figure 5.1: A hypertext link where the destination is a point or region within a document

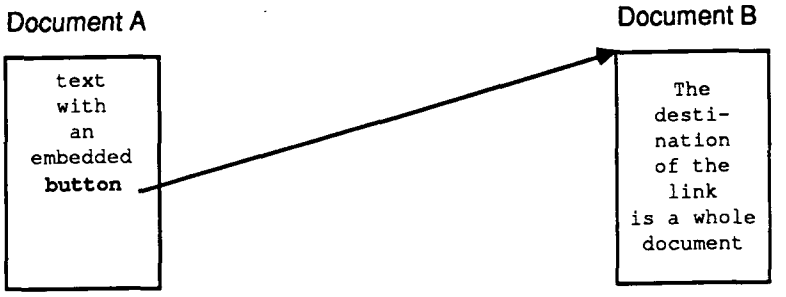


Figure 5.2: A hypertext link where the destination is a document

within the destination document. Many of the hypertext systems that have been discussed in the literature do not allow the destination of a link to be part of a document as shown in Figure 5.1. They only allow a link to point to a document (Figure 5.2).

HyperCard does not meet either of the requirements because, firstly, although it is commonly referred to as a hyper*text* tool, it does not directly support the placement of textual buttons within fields of text. Secondly, link destinations are only allowed to be cards and not points within cards. Since Guide meets both requirements, is easily obtainable and runs on relatively cheap hardware, this system was chosen for the initial investigations. There are research hypertext systems like Neptune[25] (See also Section 3.3.2) and Intermedia[64] that would have been more suitable as they offer concurrent multi-users and better management of large hypertext networks. These additional features would be necessary in a practical system for documenting software but as a medium for demonstrating initial ideas, Guide has proved adequate.

Guide was initially a research project at the University of Kent[14] running under UNIX, but has since been developed by Office Workstations Ltd. (OWL) as a commercial product for

the Apple Macintosh and the IBM PC. For the research here, it had originally been hoped to use a version of the UNIX Guide running on a Sun Workstation, but it was discovered that the UNIX version does not provide the conventional, non hierarchical hypertext link which allows jumps to different points in the document or other documents. Brown[15] has since discussed why this type of link was not provided in UNIX Guide. The main advantages of the UNIX version are that it runs on a large screen workstation and, since the system is based on the UNIX file store and the structure of Guide documents is freely available, programs can be written to automatically create Guide documents.

## 5.5 Structure of the Proposed Documentation Hypertext

The DOCMAN system comprises the following five entities:

1. Source Code
2. Cross-Reference Listing
3. Encyclopaedia Documentation
4. Glossary Documentation
5. Overview Documentation

Figure 5.3 shows schematically how these five entities are linked in the proposed hypertext redocumentation system. The following links are provided:

- Each identifier within the source code is made a button which is linked to its corresponding cross-reference entry (link type a).
- References to identifier usages in the cross-reference entries are made into buttons that are linked to the point in the source code or the documentation where the particular usage occurs (link types b and c respectively).
- Each cross-reference entry has a 'description' button that links to its corresponding encyclopaedia entry (link type d).

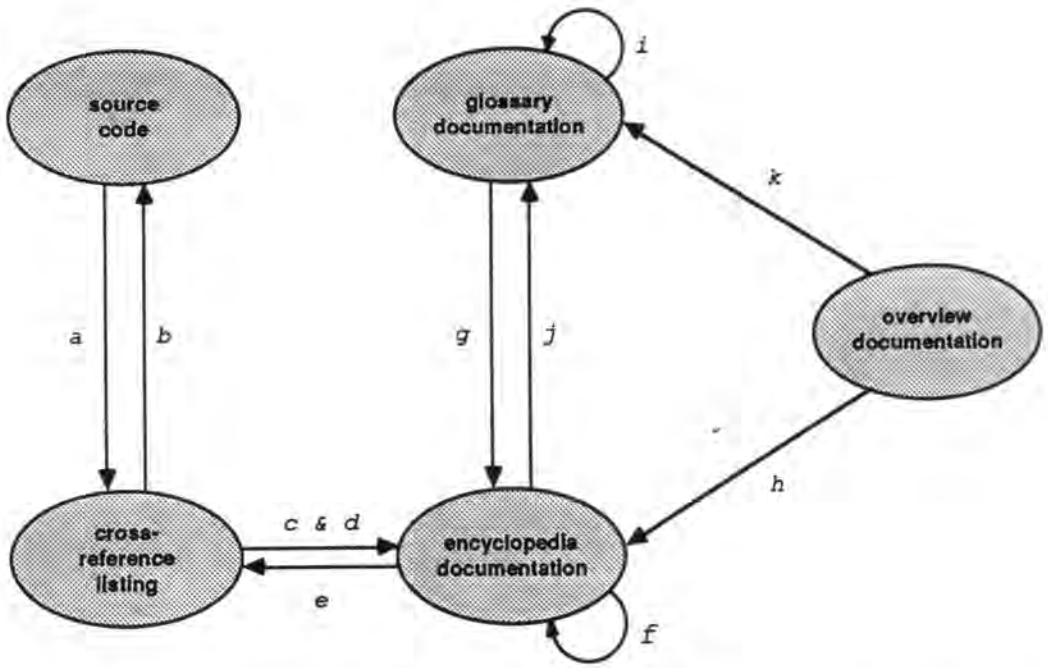


Figure 5.3: Schematic diagram of source code browser and documenter

- Each encyclopaedia entry has a reference button that links to its corresponding cross-reference entry (link type e).
- Identifiers used within any of the three types of documentation: encyclopaedia, glossary and overview are buttons that link to the encyclopaedia entries for those identifiers (link types f, g, and h).
- Glossary terms used within any of the three types of documentation: encyclopaedia, glossary and overview are buttons that link to the glossary entries for those glossary terms (link type i, j and k).

This research has concentrated on incorporating the first three DOCMAN entities into Guide. Of the remaining two entities, the glossary documentation is of a similar form to the encyclopaedia documentation and would therefore be treated in a similar way, however, the incorporation of overview documentation will require further research since there are problems associated with producing this type of documentation using hypertext. These problems will be discussed later.

Of the three entities incorporated into Guide: the source code is, of course, available; the cross-reference listing is generated by running XCC on itself; and the encyclopaedia docu-

mentation is created manually following examination of the source code by a programmer. Figure 5.4 shows an example of the hypertext links created between the three entities for the arbitrary identifier **page_count**. BNF descriptions of these entities are given in Appendix C.

Initially it had been hoped to produce a program that would take as inputs the source code and the cross-reference listing of XCC or for that matter, any C program, and produce a set of Guide documents with the links between the source code and cross-reference tables created automatically. Unfortunately, this could not be achieved because the structure of Guide documents is not published.

Since the links could not be created automatically, it was necessary to create the links manually using the menu commands provided in Guide. To have done this for the complete XCC program would have involved creating around 30 000 links. This was obviously impractical, and therefore the links were only created for a section of the source code large enough to demonstrate the approach. Nevertheless, it took many hours and many mouse operations to create the links.

Figures 5.5–5.7 show how each of these entities appear on the screen. Figure 5.5 shows a window containing a source code module of a program that is being examined by a programmer. Each identifier within the source code is highlighted in bold font, which indicates that the identifier is a button. If the programmer is interested in a particular identifier they can select the appropriate button with the mouse. In this example the programmer has selected the identifier **p_token** from the source code which causes a second window(Figure 5.6) to be opened containing the cross-reference entry for that identifier. Within the cross-reference entry there are references in the form of pathnames that uniquely describe the position of every occurrence of that identifier in both the source code and the documentation. Each of these references is also a button that is linked to the point in the source code or documentation where the identifier actually occurs. By making use of these buttons a programmer can efficiently move between points in a program and its documentation where a particular identifier is used.

Each cross-reference entry contains a button named DESCRIPTION that is linked to the encyclopaedia entry for that identifier. A similar link is provide from the encyclopaedia

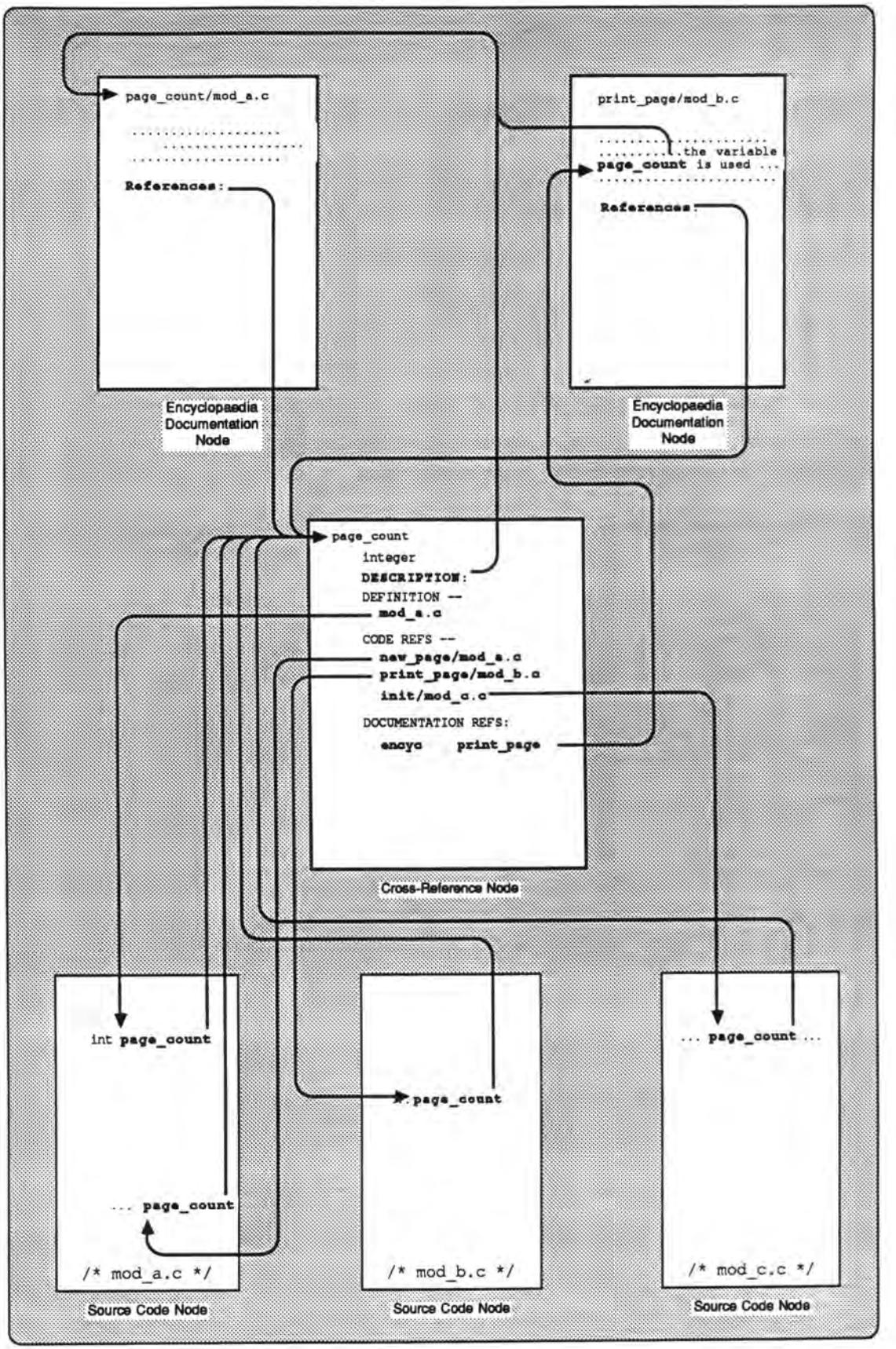


Figure 5.4: An example of the links created in Guide between the source code, cross-reference tables and encyclopaedia documentation

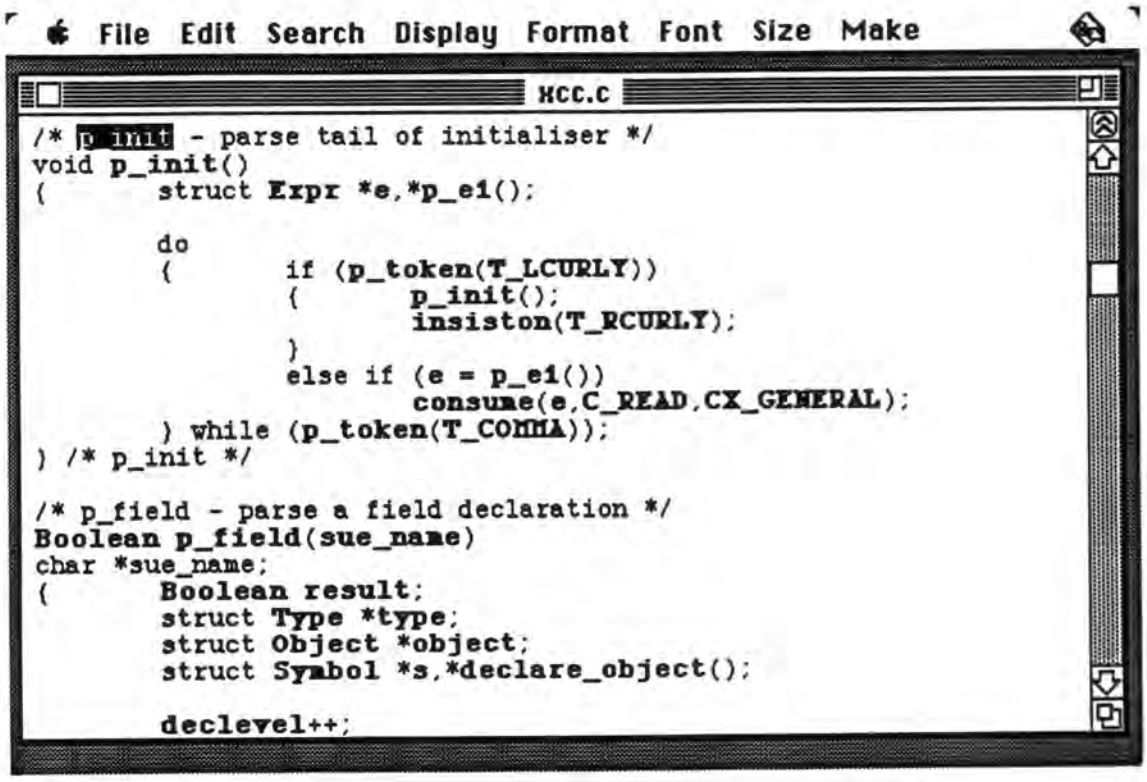


Figure 5.5: Source Code Window

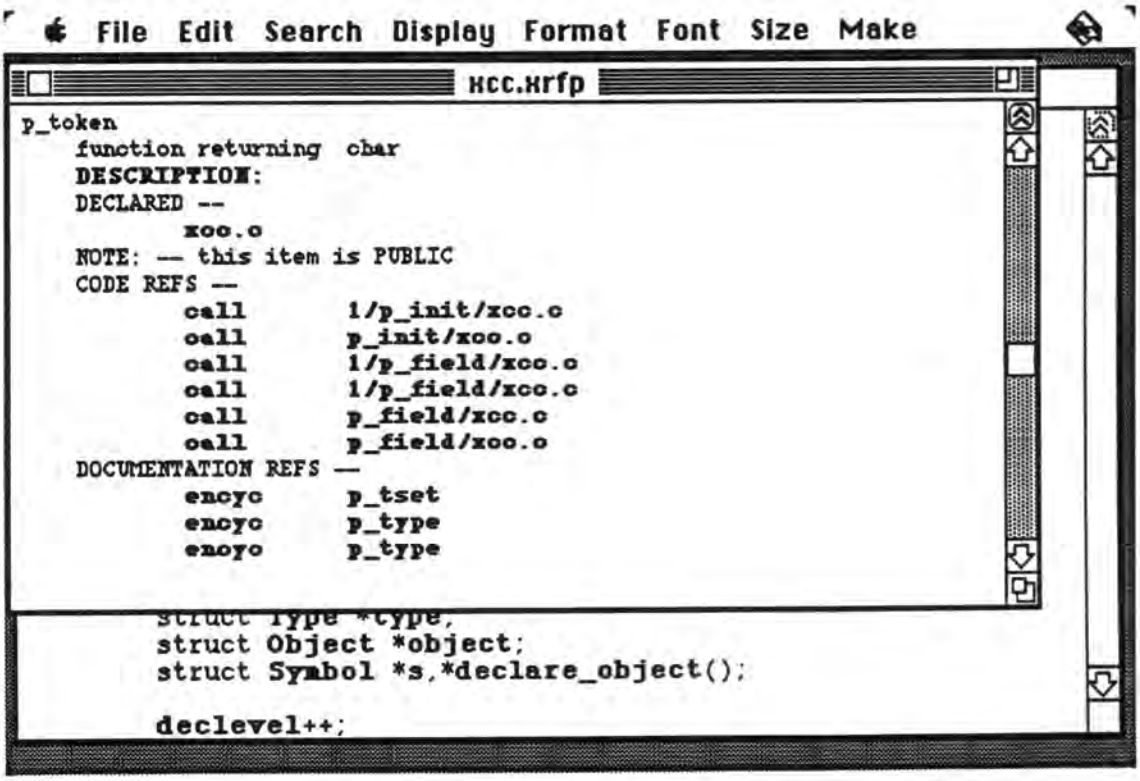


Figure 5.6: Cross-Reference Window

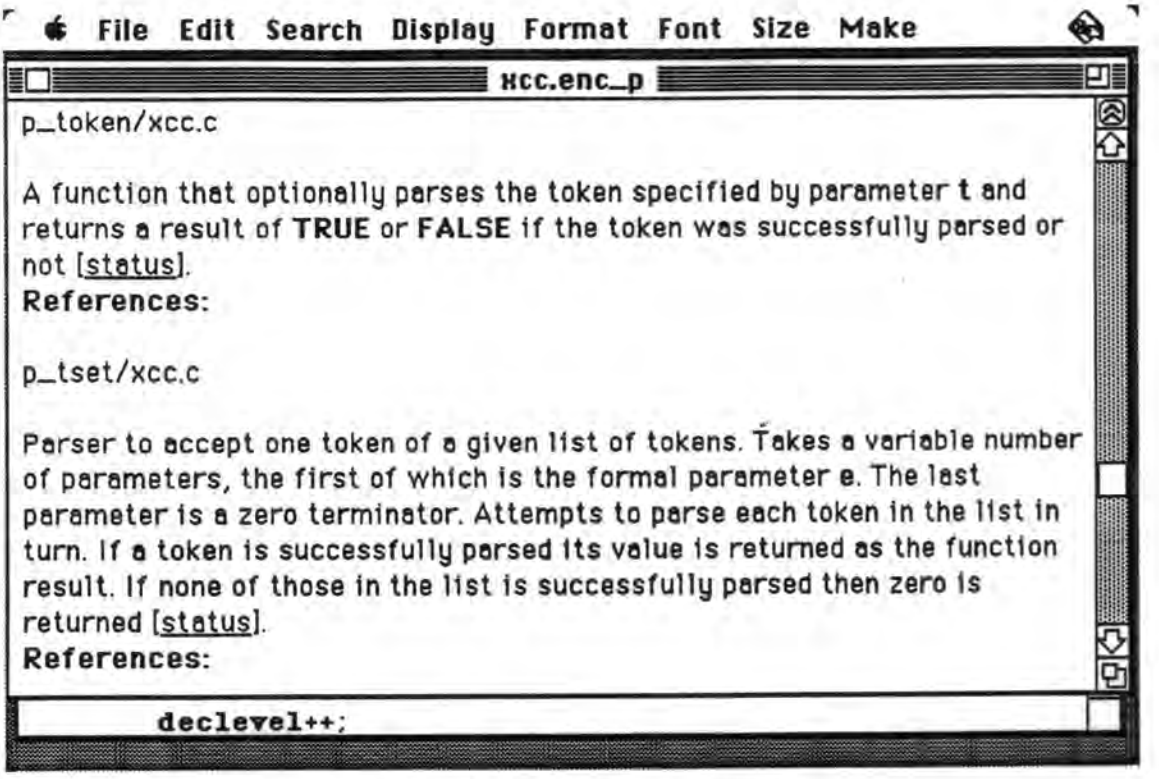


Figure 5.7: Encyclopaedia Window

entry back to the cross-reference entry by the REFERENCES button. Figure 5.7 shows the window that is created when the programmer selects the DESCRIPTION button in Figure 5.6. The new window contains a textual description of the identifier **p_token**. The description may also contain identifiers which are also buttons that point to their corresponding encyclopaedia entries. Within the encyclopaedia entry an example of a Guide note button can be seen. The button is labelled '[status]' and has been used to indicate the author and status of the description it follows.

## 5.6 How the System Would be Used

The proposed browsing and documenting system would be used by a maintenance programmer whilst examining source code during maintenance activities.

### 5.6.1 Locating Identifier References

The system will enable a programmer to locate any reference point of an identifier efficiently. An example follows of a typical task carried out by a maintenance programmer and a comparison is made between the approach that would be used with and without the proposed system.

Often, when examining the source code of a program, a programmer will wish to examine the definition of a routine that is used at the point in the code currently being examined. If the programmer does not know where the definition is located in relation to all the modules that comprise the program, it will be necessary to find the location from a cross-reference listing or by performing a global textual search on all the modules. Then when the module containing the routine has been located, it must be loaded into and editor for examination. Although this type of movement from module to module is a common one, it will have taken the programmer several minutes to complete. In which time, the break in concentration may well have caused the original reason for examining the routine to have been forgotten.

With the proposed system, a programmer using a mouse device would select the identifier of the routine at the point in the code where its use is of interest, causing the cross-reference table for that identifier to be displayed in a window. In this example the programmer would select the reference in the table associated with the definition of the routine which in turn would cause a window to be opened displaying the module at the point where the routines definition occurs. A process of only two steps that will have taken only a few seconds to complete. Likewise, a programmer can locate any reference to an identifier in the source code or documentation by two similar steps.

### 5.6.2 Creating Documentation

Initially, when the source code of a program is loaded into the system there will be no encyclopaedia documentation present. As the maintenance programmers examine the program in attempting to understand its operation, they will gain knowledge. This is when the encyclopaedia documentation is created. The mechanism must be efficient for creating the documentation otherwise programmers will be reluctant to record the knowledge they

have gained.

Encyclopaedia documentation is suitable for the maintainer to produce because it can be created with limited knowledge about the program as a whole. This is particularly important as much of the maintenance work on large programs is performed with only localised knowledge of the software in the area of a change[45].

## 5.7 Results

This section discusses the results from using Guide to demonstrate ideas for a source code browsing and documenting system.

### 5.7.1 Large Screen

The size of the Apple Macintosh screen effectively only allows one window to be viewed at any one time. A large workstation screen would offer several advantages for the redocumentation system. It would allow several windows to be viewed in parallel and windows to be positioned in fixed parts of the screen according to their role.

### 5.7.2 Encyclopaedia Entries

With the hypertext structure used for this study (Figure 5.4), the encyclopaedia entry for each identifier is a separate entity that is displayed when the programmer selects the DESCRIPTION button in the cross-reference entry for that identifier. A programmer therefore has to make two selections to open an encyclopaedia entry from the the source code. Firstly, selecting the identifier in the source code and then selecting the description button in its cross-reference entry. It would be more efficient and logical to display the encyclopaedia entry, if present, with the cross-reference entry. In this way the encyclopaedia entry and the cross-reference table of an identifier are always displayed at the same time. This can be achieved by either embedding the encyclopaedia entry within the cross-reference entry or

by having a separate encyclopaedia entry window that is automatically updated when the cross-reference entry changes.

An encyclopaedia entry will typically contain a few sentences of text describing the use of an identifier in the program. It is important to avoid repeating information in the entry that can easily be obtained by looking at the code or other encyclopaedia entries. For example it might be tempting to give the range of an array identifier or in the entry for a routine name, information could be given about the parameters to the routine. In the first case this information is directly available in the source code from the definition site of the array and in the latter case the information for parameter information should be in the encyclopaedia entries for the parameters.

### 5.7.3 Window Sizing

Within Guide there is no control over the size of newly created windows. For the application here windows need to be sized according to their contents. For instance a source code window needs to be large enough to display a minimum of around twenty four lines of code (typical size of a normal terminal). While a window containing an encyclopaedia documentation entry can be much smaller since a typical entry only spans approximately ten lines of text.

### 5.7.4 Window Creation

Guide allocates a new window to each document as it is opened until the maximum number of windows has been created. Once this point has been reached, Guide will not open any more documents until an existing document in a window has been closed. For this application and on a workstation with a larger screen it would be better to have a fixed number of windows. Each window would be allocated a certain type of document that it can contain. When a document is opened, instead of creating a new window, an existing window of a type that matched the document would have its contents replaced by the new document.

### 5.7.5 Command Language Interface

It would be extremely useful if the hypertext system had a command language interface as well as a menu driven interface to provide access to more powerful commands and the ability to run a script of commands from a file to perform repetitive or frequently used sequences of commands. Example of commands that would be useful are:

- Often a maintenance programmer will not remember the exact name of an identifier that is of interest. String searching commands that match against regular expressions would help in locating the identifier. It would be possible to dynamically create a list of buttons that are linked to the cross-reference entries of the identifiers that matched the search in a similar way to that achieved in the Symbolics Document Examiner[63].
- When creating and modifying the hypertext it is inevitable that buttons and reference points will become detached. i.e. a reference point will have no button linked to it or a button will have no valid link to a reference point. It would be useful to have a command that lists such buttons and reference points.

### 5.7.6 Flagging of Unusual Code

Often, while analysing source code during software maintenance a programmer is confronted by a section of code that appears unusual in some way. It may appear erroneous or perhaps it may appear that the input data will never cause a particular path of a program to be executed. Usually as a more extensive understanding of the system is achieved the purpose of these sections becomes clear. However, in some cases the initial hypothesis is confirmed by more detailed analysis. If the defective section of code is functionally removed from the area of the maintenance change currently being worked on, then it is common practice when dealing with large systems to report the problem as a defect for a further maintenance change.

When browsing the source code in the hypertext environment it would be useful to have a mechanism where a programmer could flag (with comments) suspect sections of code for further investigation. If, as a more complete understanding of the system is established,

the initial assertion proved false then the flag can easily be removed. But, in cases where the assertion is confirmed the flag acts as a pointer to defective areas of code to help other programmers and to mark where further attention is required.

### 5.7.7 Credibility Rating for Programmer Hypotheses

During the analysis of source code a programmer makes hypotheses about the functioning and purpose of items in the source code. These hypotheses are later confirmed or refuted by further analysis of the system as a whole (need reference here). This process is usually a purely mental process, however there could be a case for the programmer recording these hypotheses if the recording mechanism is sufficiently fast to avoid hindering the process of comprehension. To achieve this, it would be possible to provide some mechanism for the maintenance programmer to attach a credibility rating to the encyclopaedia documentation to allow both hypothesis and fact to be recorded in the entries.

### 5.7.8 Automate Creation of Encyclopaedia Links

It was found while writing encyclopaedia entries in Guide that creating the links between buttons in entries and other entries was time consuming and interrupted the flow of thought. Therefore it would be necessary to improve on the link creation methods offered in Guide for a production redocumentation system. Since, in this application, the buttons will always be linked to either an encyclopaedia entry or a glossary entry, it would be possible to partially automate the creation of these links. One possible implementation to achieve this would be for the user to select the word or phrase to be made into a button, then the system would offer the user the choice of creating a link to the encyclopaedia or glossary entries that matched the word or phrase via a menu.

### 5.7.9 Accessibility of the Documentation

A major advantage of the approach used here for browsing and documenting programs is the ease with which the documentation can be created, updated and examined by a programmer in parallel with examining the source code. Using Guide has demonstrated this advantage, but to exploit the full potential of the approach a specialised hypertext tool is required for this application.

### 5.7.10 Efficient Location of Identifier References

As the example in Section 5.6.1 illustrated, this approach offers efficient location of any occurrence of an identifier in the source code or the documentation. From a survey of documentation tools, it would appear that no commercial software tool offers similar capabilities.

### 5.7.11 Management of Large Document Sets

A problem encountered with Guide for this application is that it offers no facilities for the management of large document sets. All the documents that form part of a hyperdocument must be in the same directory. Where a hyperdocument consists of hundreds of separate documents, as is often the case in this application, the management of the documents soon becomes a problem. Some form of librarian system is required to remove this responsibility from the user.

### 5.7.12 Navigation

Guide provides a backtracking facility that allows links to be closed in a reverse sequence to that opened. By using backtracking a user can return to a location in the hypertext back down the navigation path. This facility was not found necessary in this application. The well defined structure of hypertext and the range of links available to the user at any point in the hypertext made this facility redundant.

The 'disorientation problem' is quite common in hypertext systems and consists of two problems[24]:

(a) Knowing where you are in a network.

(b) Knowing how to get to some other place in the network.

These problems were not encountered in this application of hypertext. The reason for this is likely to be the same as the reason for the redundancy of backtracking explained above.

Although backtracking was not required and the disorientation problem was not experienced with the DOCMAN entities used in this experiment, the same may not apply if the system were to include overview documentation. Unlike the other DOCMAN entities, and in common with many other hypertext application areas, the information presented in overview documentation does not have a well defined structure.

# Chapter 6

# A Prototype Source Code Browsing and Documenting System

## Introduction

This chapter discusses the important design issues encountered when developing a prototype for a hypertext based source code browsing and documenting system to form part of the DOCMAN suite of programs. The requirements for the prototype having been established from the groundwork performed by using a commercial hypertext tool, Guide, for the same application (Chapter 5).

The prototype forms the foundations for a specialised system incorporating the capabilities and ideas that had been investigated using Guide. Time constraints made it impossible to develop a system which shared the features of the generalised hypertext technology; for instance the interactive creation and manipulation of text and buttons; and, in addition, included those features identified as important for a specialized source code browsing and documenting system. Such a system would require several man-years of effort. Therefore

the prototype focuses on the implementation of features that could not be investigated in Guide.

## 6.1 DOCMAN and Cross-Referencing

DOCMAN is a documentation system based on cross-referencing developed to meet specific problems encountered in maintaining a software system of several hundred thousands of lines of code (Section 3.3.1). The left hand section of Figure 6.1 shows the existing cross-reference components of the DOCMAN suite of programs. It shows the three phases required to produce a paper or machine readable cross-reference listing for a program:

**Source File Processing** Each compilation unit is processed by a 'front-end' program designed to interpret the source language in which the compilation unit is written in. The output from each front-end is an intermediate file in a common, language independent format.

**Merging** The intermediate files produced by the previous phase are merged into a single file in the same format.

**Formatting** The final phase formats the contents of the intermediate file into a readable form; either, plain text for machine reading and low quality printing, or LaTeX where high quality printing is required.

## 6.2 Capabilities of the Prototype

As the capabilities of the prototype had to be restricted because of the time available, it was decided to investigate two areas that had not been looked at previously because of limitations in the version of Guide available and the hardware that it runs on.

When using Guide, it had not been possible to create the links between identifiers in the source code and the cross-reference tables automatically. The reason for this is that Guide

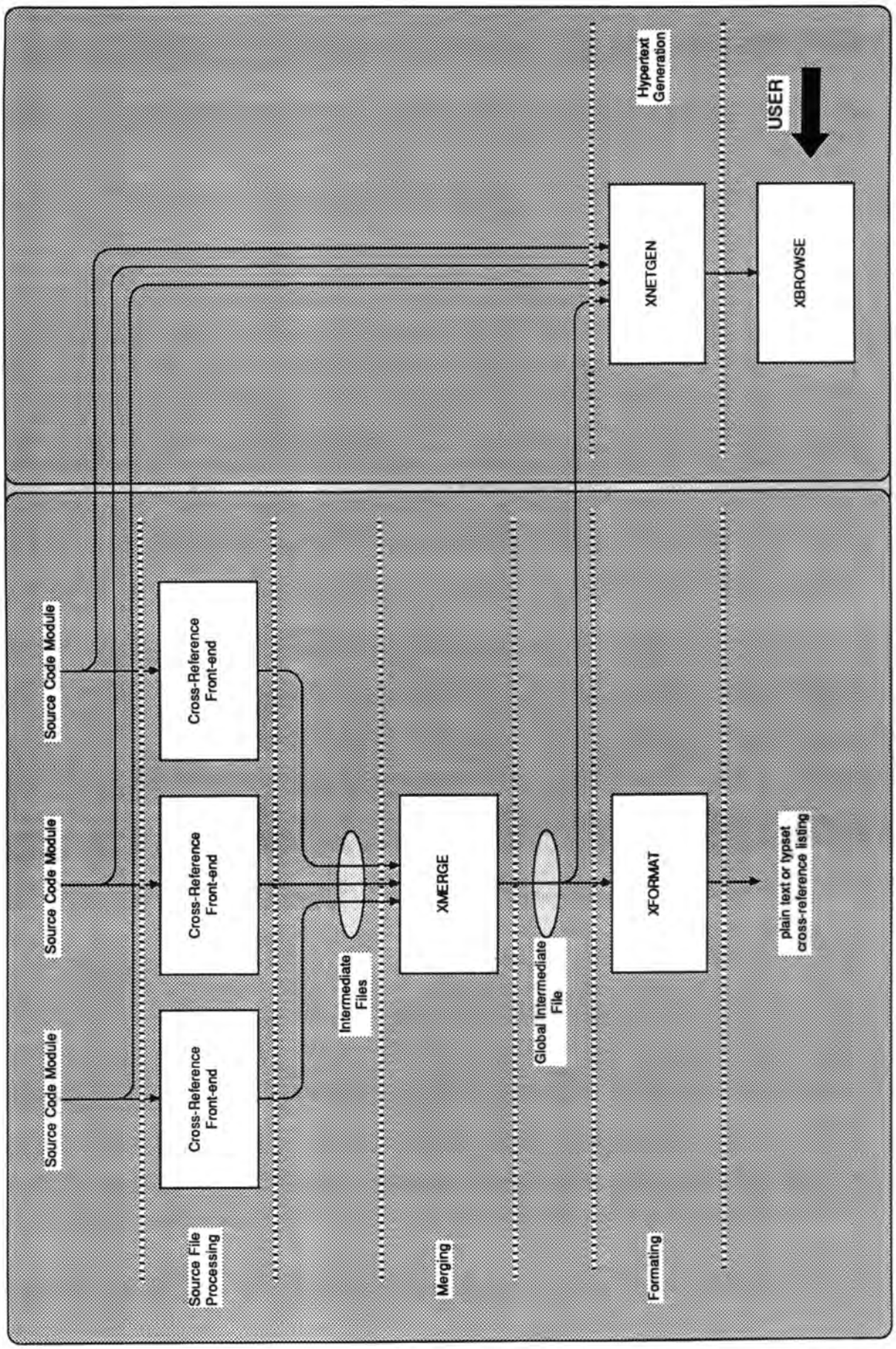


Figure 6.1: Cross-referencing part of the DOCMAN system and the extension provided by the prototype.

only allows users to create links interactively via pull-down menu commands. It does not allow structured documents to be imported into Guide with links created automatically according to the documents structure. This limits Guide to applications where documents are authored within it. An example of an application where hypertext has been used for displaying structured documents which were not prepared as hypertext documents, is the conversion of UNIX man pages into documents that can be displayed using the UNIX version of Guide[16]. Systems with built in programming languages, for example the language HyperTalk in HyperCard[36], provide the user with the capability to extend and tailor the system in this direction.

Another area where Guide proved restrictive was the small screen size of the hardware that it runs on in relation to the large screens that are now available on workstations. This limits the user to viewing only one window at a time and prevented any experimentation with window layout and window allocation algorithms. The prototype makes use of the large screen available on the Sun workstation.

The prototype has no editing capabilities and therefore it is not possible to create the encyclopaedia documentation as it had been in Guide. It is purely a system for browsing the hypertext structure created automatically between the source code and the cross-reference tables. Cross-referencers may be considered as a tool to assist programmers in navigating around a software system. The prototype browser improves their effectiveness.

The prototype offers an alternative way to view the cross-reference information generated by DOCMAN.

A new phase, 'hypertext generation' (right hand section of Figure 6.1), has been added to the DOCMAN system. This takes as input the merged intermediate file and each of the compilation units and produces a special set of documents. These documents consist of the source code and cross-reference tables in a hypertext format (see Appendix B) that can be viewed by another program, XBROWSE, running on a Sun Workstation. Figure 6.2 shows an example of the links created automatically by the hypertext generator program between the source code and the cross-reference tables. By referring to Figure 5.4 on page 65, which shows the hypertext structure used in Guide, a comparison can be made between the links that were created manually with Guide and those created automatically by the hypertext

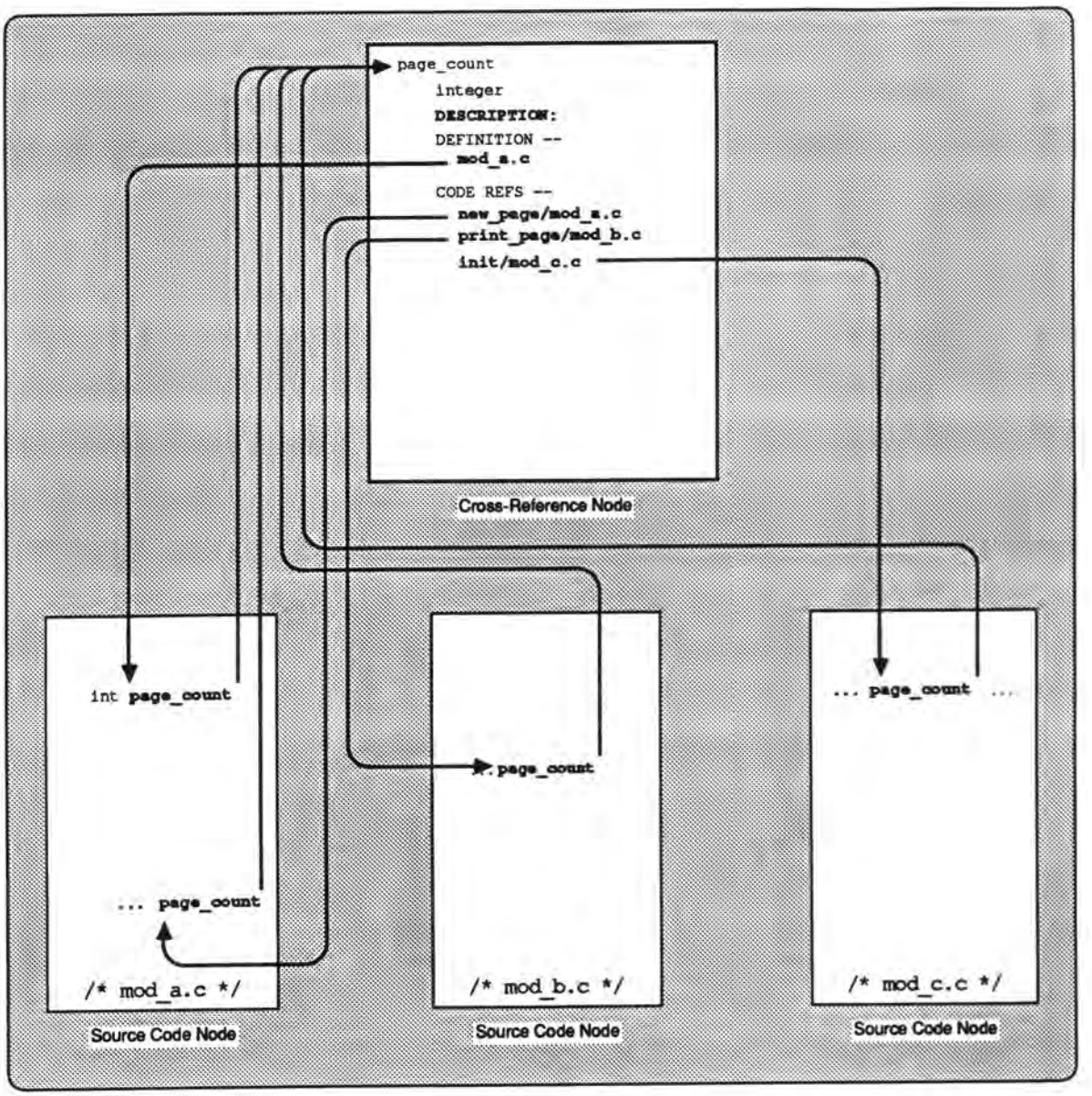


Figure 6.2: An example of the links created automatically, by the hypertext generation phase of the prototype, between the source code and the cross-reference tables

generator.

The prototype is split into two components. The first component generates the source code and cross-reference hypertext and the second component allows the user to navigate around this hypertext using a mouse and a window environment.

## 6.3 Hypertext Generation

The source code and cross-reference hypertext is generated by a program called XNET-GEN. This program takes as input the source code for a program and the intermediate file produced from this source code and generates as output a set of documents that represent the nodes of a hypertext. Each node contains either source code or cross-reference table entries.

### 6.3.1 Links

There are a number of ways to implement links between the nodes of a hypertext. The properties of the links in the prototype and the justification for them are discussed below:

1. In XBROWSE there is only a single link type that allows a user to jump from point to point in documents. Many hypertext systems have a predefined or user defined set of link types. The main purpose of typing is to provide the user with information about the destination of a link without having to open the link. For instance, a link in a document expressing an opinion may have a type of 'supports' to indicate that the linked document provides support for the opinion. Users of XBROWSE do not need this facility since the destination of a link is implicit from its context.

2. A links destination is a point within a document with no embedded text. All destination points in the hypertext are identifier names in the source code or cross-reference entries. Ideally the destination points would be the text string of the identifiers, but these strings will also be embedded in the link buttons (property 4). Therefore, for the sake of simplicity, the destination point of a link is immediately in front of the identifier that it points to. This avoids the technical problem of making the same piece of text a button and a destination point, while being equally effective.

3. The method for defining a destination point of a link within a document, allows the document to be edited without disconnecting links that terminate in the document. They may be called 'floating' destination points. The problem can be illustrated by considering a hypertext system where the links are implemented by using a line

number and character position to locate the destination point. If a document in such a system is edited, then every link that terminates at a point in the document following the edit, will need updating. Since the prototype is only a browsing system and does not have editing facilities, this property is not necessary. But, to allow future development of the prototype, floating destination points have been implemented in the system.

4. A button that marks the source point of a link in a document is a region of text, at least one character in length. This region does not cross line boundaries.

   In this system, the region of text is either an identifier name or a cross-reference path-name. This text needs to be embedded in the button as it will be the mouse sensitive area that the user selects to open a link. As neither of these two possible button text strings cross line boundaries it is therefore not necessary for the text embedded in buttons to extend over multiple lines in the prototype. The same will apply if the system is extended to include encyclopaedia and glossary documentation. However, if the system were to include overview documentation then it may be necessary to reconsider this position.

A links source structure consists of the text forming the button, the name of the document where its destination is, and a destination point key that uniquely matches with the key of a destination point structure in the destination document. The document name and key provides enough information to for a program to locate the destination of a link. The destination structure only needs to contain a key that matches with the key in the source structures.

The internal structure of the source and destination points of a link can be examined by referring to the syntax of XBROWSE documents in Appendix B.

## 6.4 Hypertext Browser

The source code and cross-reference hypertext generated by XNETGEN is viewed by the hypertext browser program, XBROWSE. This section discusses the design of the browser.

### 6.4.1 Operating Environment

The prototype browser is designed for a Sun Workstation running the SunView environment[59, 60]. The choice of workstation and window environment was based on availability only. Any large screen workstation with a window environment would have been equally suitable for demonstrating the concepts. An alternative window environment which through future standardization may offer portability of applications amongst workstations, is the X Windows system[53]. Any continued development of the prototype would benefit from the use of a standardized environment, when one becomes available.

### 6.4.2 Screen Layout

The hypertext created by XNETGEN comprises nodes that contain either source code or cross-reference entry documents. These nodes are displayed on the workstation screen in scrollable windows. Each window has a designated type which determines the type of document that can be displayed in it.

Figure 6.3 shows the default window layout for XBROWSE. The top left window allows the user to list and load the hypertext documents in the current directory. The other four windows are for displaying documents. The bottom two windows have been configured for displaying source code documents and the smaller two windows at the top right of the screen, for cross-reference entries.

The document windows are similar in appearance to those in the Guide system, although in the prototype they make use of the larger screen available on a workstation. The vertical scrollbar in the document window allows the user to browse through the document and gives an indication of the size of the document.

When XBROWSE starts, it sets the number of windows, their position and size according to data read from the SunView defaults database. Making use of the database allows the initial configuration of the screen to be adjusted by the user to match their personal preferences, without the need to recompile the browser.

Figure 6.3: Example layout of XBROWSE screen

Window resizing and repositioning while the browser is running is handled by the SunView environment. The application is only responsible for updating the image in the window following a resize. No commands have been provided to increase the number of windows displayed on the screen from the number setup at initialization; although their construction can be easily achieved.

### 6.4.3 Window Typing

The windows for displaying the nodes of the source code and cross-reference hypertext are typed. A document can only be displayed in a window of a matching type or the type 'General'. The following window types are valid in the prototype:

(i) **Source** — The source code of the program.

(ii) **Xref** — The cross-reference tables generated from the program

(iii) **Encyc** — Encyclopaedia documentation.

(iv) **Glossary** — Glossary documentation.

(v) **Overview** — Overview documentation.

(vi) **General** — A general purpose window that may display any document type.

Only types (i), (ii) and (vi) are currently used in the prototype.

### 6.4.4 Window Allocation

The algorithm used for displaying the document at the destination of a link attempts to find a window that satisfies one of the following rules in the order given:

Rule i: A window that already contains the destination document.

Rule ii: The first window that is not displaying a document and whose type matches that of the destination document.

Rule iii: The least recently used window whose type matches that of the destination document.

Rule iv: The first window that is not displaying a document and whose type is 'General'.

Rule v: The least recently used window whose type is 'General'.

If none of these rules are successful at seizing a window, an error message is displayed and the destination document is not displayed.

The least recently used window is the window with the longest elapsed time since a user command was directed at it. Commands that cause the least recently used timer of a window to be updated in the prototype are: document scrolling, button selection and traversing a link that terminates in the window.

A user of the browser may wish to have a window permanently assigned to a document for part of or the whole of a session. Thus preventing the window from being used for the display of another document. Window locking will provide this capability by the window allocation algorithm ignoring windows that are locked.

### 6.4.5 Hypertext Links

The properties of the links in the prototype browser and the justification for them are discussed below:

1. The text embedded in a button is emphasised by emboldening to indicate to the user where the mouse sensitive areas are for opening links.

2. When the user navigates around the hypertext, from document to document, the destination point of a link within a document needs to be emphasised on the screen. Without this, the user will have to scan through the part of the document visible in a window for the text string that matches the link just traversed. This is not a problem with cross-reference entries as the destination points are always positioned at the top of the window. The same would apply to encyclopaedia and glossary entries.

However, in source code the destination point may be located at any position in the window (property 3). In many instances there will be several occurrences of the same identifier in the window. The user will be unable to determine which is associated with the link just traversed.

The prototype does not implement emphasis of the destination point. Nevertheless, this is an important property and should be implemented in any future development.

3. The prototype browser always positions a destination point at the top of a window. This approach is appropriate for cross-reference, encyclopaedia and glossary document entries since the user will always want to have as much of these entries in view as possible and this is best achieved by positioning the destination point at the top of the window.

   For source code documents the positioning of the destination point requires more consideration. From using the prototype it would appear that different strategies are required depending on the properties and context of the object that a link points to in the source code and the status of the destination document at the time the link was traversed. Several points have to be considered:

   - If the object pointed to in the source code is the definition of a routine or a formal parameter of a routine then the user will want to view as much of that routines definition as possible. Therefore, in this instance the destination point (object) should be positioned at the top of the window. The only problem here is that if the routine had leading comments then these would be positioned off the top of the view.
   - For all other types of object, including definitions and usages of variables, routines, constants, etc., two different strategies are required for positioning the destination point. If the point is already in an existing view then destination point should be emphasized and the view should not be scrolled. This strategy assumes that the user will have been working recently with the window and will be happy with its content. To scroll the view to reposition the destination point would prove distracting. A second strategy is necessary when the destination is not in an existing view. The best approach here would seem to be to centre the destination point in the view.

More complex approaches could position the start of the enclosing code block of the destination point at the top of the view provided the destination point would still fit in the view. For example, in the following code fragment, if the destination point is the identifier count the code would be positioned with the line that reads { if (rp_id_count >= NUM_RP_IDS-1) at the top of the window. To maintain programming language independence in the browser and documenter the additional information required for positioning the destination point would have to be contained in the hypertext link.

```
print_char = TRUE;
while (line[i] != '\0' && actual_cp < n)
{   if (line[i]==REFERENCE_POINT_ID)
    {   if (rp_id_count >= NUM_RP_IDS-1)
        {   rp_id_count = 0;
            print_char = TRUE;
            ++count;
        }
        else
        {   ++rp_id_count;
            print_char = FALSE;
        }
    }
    else if (line[i]==REFERENCE_BUTTON_ID)
    {   if (rb_id_count >= NUM_RB_IDS-1)
        {   rb_id_count = 0;
```

### 6.4.6 User Commands

The user interface to the prototype is very simple. Browsing of the source code and cross-reference hypertext is performed either by scrolling through documents using the scrollbar or by locating the mouse cursor over a button and pressing the left mouse key. In a production version of the prototype a wider range of commands would be needed to provide alternative ways of locating areas of interest in the hypertext. But the basic browsing commands, as

provided in the prototype, would remain the most used.

### 6.4.7 Response Time

It is widely recognised that the most important requirement of any hypertext system is a rapid response between the user selecting a button and the display of the document associated with the link. Many implementation factors will effect this response time.

A response time of 1 to 2 seconds is considered the maximum acceptable for a hypertext system[24]. Any longer and their is a risk that the user will become distracted during the wait. It would be expected that the shorter the time the better, however, experience with the ZOG system running on a machine capable of response times in the range 0.05 to 0.1 seconds showed that at the lower limit of this range, users had trouble detecting whether or not the screen had changed[3]. The provision of some alternative cue, such as emphasis of the destination point for a fixed time period, would overcome this.

No measurements have been made of the response time of the prototype, but most responses appear to be in the sub one second range. Times could easily be collected by modifying the prototype to collect usage statistics.

The prototype uses document caching to reduce response time by taking advantage of the large local memory available on workstations. During a session with the prototype once a document has been opened it will remain in a cache while there is sufficient memory. When a fresh document is opened and there is no available memory in the cache then the least recently used document is removed to make room for the new document. The source code and cross-reference hypertexts for large software systems will consist of many hundreds, if not thousands of separate hypertext documents. A programmer browsing such a system while investigating a maintenance change will usually only be examining a small part of the complete system. Therefore it is likely that during a browsing session the set of documents opened will be small when compared with the total set for the system. The set may be sufficiently small for the majority of documents to remain permanently in memory. Again, automatic statistic collection would enable this hypothesis to be verified.

The hypertext links in the prototype have been implemented by using a key embedded in the text of a document to mark the destination point of a link. When a link is traversed the destination document is scanned from the beginning to locate the key that matches the corresponding key of the source point of the link. The time it takes to locate the key is therefore dependent on the position of the key within the document and the size of the document. When a link terminates at the end of large document this search time can have a considerable effect on the link response time. Therefore large documents should be avoided.

To avoid one large cross-reference listing document a separate document has been created for each set of identifiers beginning with a different ascii character. Obviously, the size of the listing will be dependent on the size of the software system it was generated from. For very large systems the approach taken here may not be adequate and an alternative method of indexing into the cross-reference listing may be necessary.

### 6.4.8 Enhancements

It would be interesting to provide the capability for automatically collecting statistics about the use of the system. These could then be used to tailor the design of the system to provide the best performance and functionality for the users needs.

For the authoring of textual documentation, in the form of encyclopaedia, glossary and overview documentation, it will be necessary to add editing capabilities to XBROWSE. For both ordinary text and hypertext link structures.

When a software system is being maintained the code will go through many revisions. When using this browsing system is will be necessary to regenerate the source code and cross-reference hypertext following each revision. If the browser included the textual documentation components then the links created between the documentation and the cross-reference entries will become detached. A scheme is therefore required to enable these links to be regenerated automatically where possible. At the same time, the user could be alerted of areas of documentation that may need updating following the revision.

As experienced with Guide, the prototype suffers from the problem of how to manage and

organise a large document set. The prototype expects to find all the documents associated with a hypertext in one flat directory: the current directory. With a large software system this would become a burden.

## 6.5 Requirements

The requirements for a source code browsing and documenting system based on the experiences from using Guide and developing the prototype are given in Appendix A.

# Chapter 7

# Further Research and Development

## Introduction

The prototype has demonstrated an approach to documenting software systems during maintenance. It has opened up many interesting areas for further research and development. This chapter discusses those areas.

## 7.1 Inclusion of Overview Documentation

The research here has established how source code and the DOCMAN entities: encyclopaedia and cross-reference documentation can be built into a hypertext network. Two more entities exist in DOCMAN. They are glossary and overview documentation. Glossary documentation can easily be added since it is of the same form as encyclopaedia documentation and therefore can be incorporated in a similar way.

Within encyclopaedia and glossary documentation there is no confusion about what text

should be linked to where since the placement of links is well defined. However, overview documentation usually consists of free-form narrative text and the placement of links is more difficult. Strategies for positioning links in this kind of documentation is an area of active research in the hypertext community and would need to be considered.

Other areas that may be a problem with overview documentation are: the disorientation problem or getting lost, and updating the documentation following a change to the software.

## 7.2 Incremental Update of Cross-Referencing Tables

The creation of the links between the source code and cross-reference documents are created in a separate process that is performed before the hypertext can be viewed using XBROWSE. Following each update of the source code, the process of generating the cross-referencing tables and links will have to be repeated. Although this is an automated process that could be performed overnight following a days editing, it would be better if XBROWSE created the new tables and links itself following a change to the code without the need to regenerate everything from scratch. This approach would be similar to that used by incremental compilers which only recompile the units of code in a module that have changed since the last compile. A drawback of such an approach is that the tool would then become language dependent.

## 7.3 Configuration Management

One of the requirements for a redocumentation tool in Chapter 4 is that it should provide configuration management for the source code and documentation. How this will be achieved in the proposed system has not been considered here, but it is an important requirement that needs to be addressed. Two options are available. Either provide configuration management internally in the browsing and documenting system or interface the system to a separate configuration management system such as SCCS or Lifespan.

## 7.4 Static Analysis Data

Automatic documentation tools generate a large amount of information from static analysis of the source code. Much of this information is of use to the maintenance programmer, but the quantity generated can be overwhelming as no assistance is provided to the programmer for extracting relevant information. The browsing and documenting system described here has incorporated the data generated by a cross-referencer, further static analysis data may be included in the system in a similar way to provide a simple user interface to it.

## 7.5 Content of Encyclopaedia Entries

The network created by the browsing and documenting system will contain a large amount of information about the source code. The encyclopaedia entries as proposed, consist of free format text. If instead, these entries were created with a defined format, then it may be possible to use expert system techniques on the network to provide answers to queries from the programmer and to guide a programmer around the source code.

## 7.6 Team Use

Another requirement of the redocumentation tool was that it should be able to support concurrent access and update. This capability has not been provided in the prototype. In a commercial product this would be an important requirement and could be achieved by building the application around a hypertext transaction server such as the Hypertext Abstract Machine (HAM)[20] developed as part of the Tektronix Neptune system. The HAM has multi-user access built in.

Alternatively, an approach similar to that taken by KMS[3] would be possible. In this system the units of information stored in each node of the hypertext network is small. Since a typical network will be very large, users will usually be working in different areas of the network and therefore conflicts between users editing the same node are rare. On

this assumption the designers have chosen a simple concurrency control mechanism, called optimistic concurrency control, which guarantees that a user cannot have successfully saved changes revoked by another user. But, if an editing conflict does occur, then the user will not necessarily be able to save their changes without problem.

## 7.7 Webs and Paths

The concept of webs and paths demonstrated in Brown University's Intermedia system[65, 64] could be used in the documenting and browsing system.

In an Intermedia hypertext network, every link belongs to one or more webs. Only those links belonging to a currently active web can be seen by the user. This concept could be used in documenting and browsing system to provide abstract views of the source code: webs would be created in the network at different levels of detail from the code and the user would choose the level appropriate to their current task.

Paths are routes through the hypertext network. These may be useful for the documentation of multi-process software. Communication between the processes is often achieved by passing messages between them. A problem found with source code documentation for multi-process software is that the documentation is usually process based. Yet system functions are implemented across several processes. Paths could be used through the documentation network to follow the trail of system wide functions through the software. Enabling the control flow to be followed from process to process.

## 7.8 Importing Existing Documentation

Although the majority of original documentation produced during design will be of little use to software maintainers, there will be some that is useful. Therefore a way of including this documentation into the hypertext documentation network created by the redocumentation system should be provided. Optical character readers are now sufficiently reliable at reading a wide range of type faces that they now offer a means to import hard copy documentation

into the system.

## 7.9 Monitoring

An area of research in software maintenance is observing how people debug computer programs[45, 37]. The experiments that have been performed in this area have been based on small programs because of the problems of collecting and analysing the data. The redocumentation system could provide a way of collecting data about the steps people go through when debugging programs. A large part of browsing through the code would be performed by using the links between the source code, cross-reference tables and the documentation. By monitoring which links are traversed, data can be collected about what parts of the code are examined and in what sequence. The analysis of such data is a possible area of research.

A less ambitious use of the data in a commercial version of the system, would be to provide management with information on what areas of the code and documentation are examined the most during the analysis phase of software maintenance. The information generated would be used to identify troublesome areas of a program that would benefit from preventive maintenance or areas in the documentation that need improvement.

# Chapter 8

# Conclusions

The research described in this thesis has met the objectives outlined in Section 1.2. It has demonstrated a technique for redocumenting source code during software maintenance that is based on ideas first developed by a maintenance team at British Telecom Research Laboratories[33]. The extension of these ideas in this thesis are now the teams recommended approach to redocumenting software.

## 8.1 Benefits of the Approach

The major benefits of the browsing and documenting system for redocumenting source code can be summarized as:

- **Efficient browsing of code and documentation**

  The system automates the low-level tasks of a maintenance programmer when browsing source code and locating relevant documentation. Hypertext links have been used to allow the programmer to quickly locate any reference to an identifier in the source code and the documentation.

- **Notepad approach to documenting source code**

The concept of encyclopaedia documentation provides sufficiently small units of documentation that a programmer can create entries without necessarily understanding a large part of the surrounding program.

- **Simple user interface**

  Most user interaction with the system is via single 'point and click' commands using a mouse.

- **Records knowledge gained during maintenance effort**

  A major objective of this research was to establish an approach for recording the knowledge gained by a maintenance programmer during analysis of a program. The knowledge can then be used by other programmers working in the same area or by the author when working in the same area at a later date. The proposed system meets this objective.

- **Only the problem area code gets documented**

  The system allows incremental redocumentation of the source code. Only the areas of the code that are examined during analysis of the code during maintenance operations are documented. No effort is wasted in documenting code that is in a stable state and never looked at.

- **Language independent** The same techniques can be applied to any programming language. The only language dependent component is the cross-referencer.

## 8.2 Drawbacks of the Approach

Two drawbacks with the system have been identified, but they are not considered to be significant in relation to the benefits.

- **Additional material to be maintained**

  The system does increase the amount of material to be maintained. When a change is made to the code, the documentation will need updating to preserve consistency between them.

- **Large screen workstation required** For the full benefits of the system to be achieved, it needs to be implemented on a large screen workstation. This is not a problem in the scientific and engineering software communities because workstations are in use and increasing in popularity. However, commercial software is still being produced and maintained on conventional 80 column, 24 row terminals which are not suitable for supporting an application of this kind. The problems will diminish as workstations become cheaper and the technical differences between PCs and workstations merge.

## 8.3 Fulfilment of Requirements

The source code browsing and documenting system has met most of the requirements considered important for a redocumentation tool in Section 4.3. These include: incremental documentation, informal update, quality assurance, integrated source code, integrated automatic documentation and information hiding. Configuration management and team use have not been addressed in the research, but they could easily be supported by the system.

The browsing and documenting system discussed in this thesis provides capabilities currently unavailable from any vendor. The prototype developed as part of this M.Sc. has been demonstrated to many industrial visitors to the Centre for Software Maintenance. Without exception, the enthusiasm shown for it has been high. I believe it can be developed into a very successful commercial product.

# Appendix A

# Requirements for a Source Code Browsing and Documenting System

This appendix describes the requirements for a source code browsing and documenting system. They have been established from the work using a commercial hypertext system, Guide, and the development of a prototype discussed in Chapters 5 and 6 respectively. The set of requirements is not complete as the intention has been to concentrate on those areas considered important for this application. Other requirements, of a more general nature, have been left vague: especially where there are several suitable approaches that may be taken.

## A.1 Overview

The system shall provide an alternative approach for DOCMAN[33] users to view the source code and cross-reference listings associated with a program being maintained. Also, it shall allow the user to create, modify and examine documentation about routines, data items,

types and other named entities in the program.

Hypertext technology shall be used to provide machine support for the links between the source code, cross-reference listings and documentation that would usually be followed manually by the user.

## A.2 Development and Operating Environment

The nature of the system necessitates its implementation on a large screen workstation with a mouse and a window environment. The window environment should preferably be a standardized one that will allow the future porting of the system to other manufacturers workstations with minimal financial overhead. To ensure performance requirements are met the workstation should have an internal memory of at least 4Mb to enable several documents to be stored in memory at the same time.

An example of an environment satisfying these requirements would be the combination of a Sun workstation, the UNIX operating system and the X Windows environment.

## A.3 External Interfaces and Data Flow

The system shall be integrated into the DOCMAN suite of programs. It shall take as input: an intermediate cross-reference file generated from the source files that comprise a program; the source files themselves; and encyclopaedia documentation. The first time the hypertext files are generated, there will be no existing encyclopaedia documentation to be included in the hypertext since the system will not have been used before. But, later generations of the hypertext will include the documentation created by the user while browsing the source code and cross-reference hypertext.

An external interface shall be provided to the host operating system to allow commands to be run from a script and to allow existing documentation to be input into the hypertext.

## A.4 Functional Requirements

### A.4.1 Windows

#### Multiple Windows

The system shall allow the display of multiple windows on the workstation screen.

#### Window Typing

Each window shall have a type associated with it that restricts the type of documentation that it may display. Window types in this system shall be: Source (source code), Xref (cross-reference document), Encyc (encyclopaedia document), Glossary (glossary document) and General. A window with type general can display any document type. All window types shall share the same set of commands.

#### Default Window Configuration

When the system is started, the number of windows, their position, size and type shall be set according to user customisable default values.

#### Overlapping Windows

The system shall allow document windows to overlap.

If when navigating between documents, the destination point of a link is in a window that is overlapped by another window then the overlapped window shall be brought to the front.



#### Window Locking

The system shall allow a document to be locked to a particular window. This will prevent the system from replacing the document in the window with a different document.

#### Miscellaneous Window Commands

Window commands shall be provided in the system to:

- Create and delete windows.
- Reposition windows.
- Resize windows.
- Retype windows.
- Scroll the documents displayed in windows.

### A.4.2 Documents

#### Scrolling of Documents Following Button Selection

The scrolling of the destination document following a button selection shall behave according to the following rules:

1. When a new cross-reference entry is displayed in a window it shall be positioned with its top line at the top of the window.
2. When a new cross-reference entry is displayed, its corresponding encyclopaedia entry shall be displayed in a separate window unless a window of the correct type is unavailable.
3. When a new encyclopaedia entry is displayed in a window it shall be positioned with its top line at the top of the window.

4. When a new glossary entry is displayed in a window is shall be positioned with its top line at the top of the window.

5. When the destination of a link is an identifier in a source code document, then the source code shall be positioned in a window with the identifier located on the line nearest the middle of the window. Thus allowing the identifier to be displayed in context.

### Document Status

Popup windows shall provide facilities for attaching statuses to textual descriptions in the encyclopaedia and glossary documentation. Mouse sensitive symbols shall be used in the text to indicate their presence. The mouse will be used to display the status information which will consist of author, status, creation date and approved date.

This facility will provide a means of supporting quality assurance for the documentation created in the system. When a new documentation entry has been created, the system shall generate a status entry and place a button at the end of the documentation entry to which it refers. The status entry will be created automatically with the author field containing the user name of the author, the status field will initially be set to 'unapproved' and the creation date field will be set to the current date. All other fields will be empty.

All new documentation entries will be reviewed to ensure their accuracy. The reviews may occur at fixed time intervals, prior to new releases of the software, when the amount of unreviewed documentation reaches a predetermined level or at any other time determined appropriate for the project. Following the successfull review of a documentation entry, its status field will be updated to 'approved' and its approved date will be set. If a documentation entry fails review, then the entry will be removed. A replacement entry may be created at this time and review process will be repeated.

#### Document Annotations

Popup windows shall provide facilities for attaching annotations to textual descriptions in the encyclopaedia and glossary documentation. Mouse sensitive symbols shall be used in the text to indicate their presence. The mouse will be used to display the annotations which will contain user created notes about the entry to which it is attached.

#### Configuration Management

The system shall provide configuration management for documents. This will be provides either internally or by interfacing to an external configuration management system.

### A.4.3 Links

#### Legal Links

One way links shall be allowed between the document types as follows:

- Between identifiers in the source code and their cross-reference entries.
- Between source references in the cross-reference entries and the point of reference in the source code.
- Between document references in the cross-reference entries and the point of reference in the encyclopaedia documentation.
- Between identifiers in the encyclopaedia documentation and their corresponding encyclopaedia entries.
- Between glossary terms in the encyclopaedia entries and their glossary entries.
- Between glossary terms in the glossary entries and their corresponding glossary entries.

#### Emphasis of Buttons

Each item in a document that is a button shall be highlighted in a bold font.

#### Emphasis of Destination Points

Following traversal of a link by selecting a button, the destination point should be emphasised for a period of time to enable the user to see the exact point in the document where the link terminated.

#### Cursor Shape

When the mouse locator is positioned over a button its image shall change. This will indicate to the user that the cursor is positioned correctly to allow button selection.

### A.4.4 Mouse

#### Button Selection

When the mouse cursor is positioned over a button and a mouse key is pressed the document that the button is linked to will be displayed in a window compatible with the documents type. The view into the document will be positioned so that the destination point of the link is positioned according to the rules in Requirement A.4.2.

### A.4.5 User Commands

#### Pull-Down Menu Interface

The system shall provide a pull-down menu command interface.

### Command Language Interface

The system shall provide a command language interface as an alternative to the pull-down menu interface to allow commands to be typed and command scripts to be run within the system.

### Text Editing Commands

The system shall provide text editing commands in line with those available on modern interactive editors. The editing commands available in a window will be dependant on the type of document that is displayed within it.

Editing commands will be allowed as follows:

**Source Code** The initial version of the source code browsing and documenting system shall not allow editing of source code. Later versions, with configuration management facilities, will allow the source code to be updated.

**Cross-Reference Document** No editing commands shall be allowed on cross-reference documents. These documents will be created automatically and will therefore not require manual updating.

**Encyclopaedia Document** General users shall be allowed to create new encyclopaedia entries and to add text to existing entries.

Removal of text shall only be allowed by 'super-users'. This would normally occur following the review of an encycopaedia entry where it had been agreed that parts of an entry were out of date and needed removing

**Glossary Document** The same rules shall apply to glossary documents as encyclopaedia documents.

If an edited document in a window is replaced by a new document following a user action, then the user shall be given the option to save the edited document before it is removed from the window.

### Link Creation and Deletion Commands

Commands shall be provided to enable links to be created interactively between:

1. Identifiers in the encyclopaedia documentation and their encyclopaedia entries.
2. Glossary terms in the encyclopaedia and glossary documentation and their glossary entries.

All other links will be created automatically.

### Search Commands

All window types shall have searching commands for exact matching of text within a document.

Additional search commands shall be provided to facilitate the location of areas of code where the user may have some recollection of names used within the area, but cannot remember the exact names. These search commands shall include:

- Search commands that match identifiers in cross-reference entries against regular expressions.
- Search commands with 'intelligent' matching algorithms similar to those used in spell checking programs that offer a number of alternative choices to the misspelt word.

For all the identifiers that match the search expression, there shall be a button dynamically created in a temporary window that is linked to the cross-reference entry for the identifier.

### A.4.6 Performance

#### Response Time

The response time between a user selecting a button and the display of the document associated with the destination of the link shall be no more than one second.

### A.4.7 Multiple Users

The system shall provide facilities to allow multiple users to access and update documents.

### A.4.8 Glossary

**button** Buttons are highlighted, mouse sensitive strings of characters that indicate the existence of a hypertext link between the button and a point in either that document or a separate document. By clicking a mouse button when the cursor is positioned over a button causes the document containing the destination of the link to be displayed in an available window.

**link** A link connects two points in a document or separate documents. The source point of the link is indicated in the document by the presence of a button.

**popup window** A temporary window created as the result of a user action. The window lasts for either the period of the user action (e.g. the operation of a mouse key) or until a second user action.

# Appendix B

# Structure of the Hypertext Documents for XBROWSE

The following syntactic description of XBROWSE documents, uses the syntactic metalanguage defined in BS6154[11, 55].

```
xbrowse-document        =   prolog, hypertext-document ;

prolog                  =   version,
                            doc-type,
                            first-free-dest-key,
                            ".do "
                            (* prologue of an XBROWSE document *) ;

version                 =   ".vn ", integer
                            (* the version of the XBROWSE
                               document *) ;

doc-type                =   ".ty ", ("Xref" | "Source")
                            (* the type of the XBROWSE document *) ;
```

```
first-free-dest-key          =    ".ky ", integer
                                  (* the first available destination key in
                                     the document *) ;

hypertext-document           =    { ascii-character |
                                     space |
                                     reference-button-structure |
                                     reference-point-structure
                                  } ;

reference-button-structure   =    reference-button-id,
                                  button-text-string,
                                  reference-button-id,
                                  destination-filename,
                                  reference-button-id,
                                  destination-key,
                                  reference-button-id ;

reference-button-id          =    control-A ;

button-text-string           =    (* a sequence of ascii printable
                                     characters *) ;

destination-filename         =    (* name of the file containing the
                                     destination of the link *) ;

reference-point-structure    =    reference-point-id,
                                  destination-key,
                                  reference-point-id ;

destination-key              =    integer ;
```

```
reference-point-id      =   control-B ;

control-A               =   ? the ascii character "^A" ? ;

control-B               =   ? the ascii character "^B" ? ;

string                  =   (ascii-character | space),
                            {ascii-character | space} ;

ascii-character         =   symbol | digit |
                            lower-case-letter | upper-case-letter ;

integer                 =   decimal-digit, {decimal-digit} ;

lower-case-letter       =   "a" | "b" | "c" | "d" | "e" | "f" | "g" |
                            "h" | "i" | "j" | "k" | "l" | "m" | "n" |
                            "o" | "p" | "q" | "r" | "s" | "t" | "u" |
                            "v" | "w" | "x" | "y" | "z" ;

upper-case-letter       =   "A" | "B" | "C" | "D" | "E" | "F" | "G" |
                            "H" | "I" | "J" | "K" | "L" | "M" | "N" |
                            "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
                            "V" | "W" | "X" | "Y" | "Z" ;

decimal-digit           =   "0" | "1" | "2" | "3" | "4" |
                            "5" | "6" | "7" | "8" | "9" ;

space                   =   " " | new-line ;

new-line                =   ? ASCII new line character ? ;

symbol                  =   "!" | '"' | "#" | "$" | "%" | "&" | "'" |
                            "(" | ")" | "*" | "+" | "," | "-" | "." |
```

```
"/" | ":" | ";" | "<" | "=" | ">" | "?" |
"@" | "[" | "\" | "]" | "^" | "_" | "`" |
"{" | "}" | "-" ;
```

# Appendix C

# Proposed Syntax of DOCMAN Entities for Source Code Browsing and Documenting System

The following proposed syntactic description of the DOCMAN entities to be included in the source code browsing and documenting system, uses the syntactic metalanguage defined in BS6154[11, 55]. Descriptions are given of the glossary, encyclopaedia and cross-reference entities.

```
glossary-docn          =   {glossary-entry} ;

glossary-entry         =   glossary-term, new-line,
                           glossary-defn ;

glossary-term          =   string ;

glossary-defn          =   {string | glossary-term-button} ;

glossary-term-button   =   button
```

```
                                (* a button that is linked to the
                                   glossary entry corresponding to the
                                   glossary term *) ;

encyc-docn              =       {encyc-entry} ;

encyc-entry             =       identifier-name, new-line,
                                encyc-defn, new-line,
                                reference-button, 2 * new-line ;

encyc-defn              =       {  string |
                                   glossary-term-button |
                                   identifier-button
                                } ;

reference-button        =       ? the terminal "REFERENCES:" in bold
                                  font ?
                                (* the button that is linked to the
                                   xref entry corresponding to the
                                   encyclopaedia entry *) ;

identifier-button       =       button
                                (* a button that is linked to the
                                   encyclopaedia entry for the
                                   identifier *) ;

xref-docn               =       {xref-entry} ;

xref-entry              =       identifier-name, new-line,
```

```
                        {identifier-description, new-line},
                        "DEFINITION:", new-line,
                        [definition-button],
                        "CODE REFERENCES:", new-line
                        {code-reference-button, new-line},
                        "DOCUMENTATION REFERENCES:", new-line,
                        {docn-reference-button, new-line},
                        2 * new-line ;

identifier-description  =  string
                           (* a string generated by a cross-reference
                              front-end giving information about the
                              identifier *) ;

definition-button       =  button
                           (* the button that is linked to the
                              definition site of the identifier *) ;

code-reference-button   =  button
                           (* a button that is linked to a
                              specific reference to the identifier
                              in the source code *) ;

docn-reference-button   =  button
                           (* a button that is linked to a specific
                              reference to the identifier in the
                              documentation *) ;

identifier-name         =  string ;

button                  =  bold-string
                           (* a hypertext button represented on the
                              screen as a string in bold font *) ;
```

```
word                    =   string - space;

string                  =   (ascii-character | space),
                            {ascii-character | space} ;

bold-string             =   ? the non-terminal string in a bold
                              font ?;

ascii-character         =   symbol | digit |
                            lower-case-letter | upper-case-letter;

lower-case-letter       =   "a" | "b" | "c" | "d" | "e" | "f" | "g" |
                            "h" | "i" | "j" | "k" | "l" | "m" | "n" |
                            "o" | "p" | "q" | "r" | "s" | "t" | "u" |
                            "v" | "w" | "x" | "y" | "z" ;

upper-case-letter       =   "A" | "B" | "C" | "D" | "E" | "F" | "G" |
                            "H" | "I" | "J" | "K" | "L" | "M" | "N" |
                            "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
                            "V" | "W" | "X" | "Y" | "Z" ;

decimal-digit           =   "0" | "1" | "2" | "3" | "4" |
                            "5" | "6" | "7" | "8" | "9" ;

space                   =   " " | new-line ;

new-line                =   ? ASCII new line character ? ;

symbol                  =   "!" | '"' | "#" | "$" | "%" | "&" | "'" |
                            "(" | ")" | "*" | "+" | "," | "-" | "." |
                            "/" | ":" | ";" | "<" | "=" | ">" | "?" |
                            "@" | "[" | "\" | "]" | "^" | "_" | "`" |
```

```
"{" | "}" | "-" ;
```

# Bibliography


[1] Federal Information Processing Standards Publication 38. Guidelines for documentation of computer programs and automated.... Technical report, NBS, U.S. Department of Commerce, February 1976.

[2] Federal Information Processing Standards Publication 64. Guidelines for documentation of computer programs and automated.... Technical report, NBS, U.S. Department of Commerce, August 1979.

[3] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: a distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[4] Roy E. Anderson. Modular documentation: A software development tool. In *AFIPS Conf. Proc. 1981 National Computer Conf.*, pages 401–405, 1981.

[5] ANSI/ANS. Guidlines for the documentation of digital computer programs. Technical Report ANSI/ANS 10.3–1986, American National Standards, 1986.

[6] ANSI/IEEE. Software test documentation. Technical Report ANSI/IEEE 829–1983, American National Standards, 1983.

[7] P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile. Maintenance and reverse engineering: Low–level design documents production and improvement. In *Proceedings of the Conference on Software Maintenance — 1987*, pages 91–100, September 1987.

[8] Apple Computers, Inc., California. *HyperCard User's Manual*, 1987.

[9] Barry W. Boehm. Software and its impact: A quantative assessment. *Datamation*, pages 48–59, May 1973.

[10] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[11] British Standards Institution, 2 park St., London W1A 2BS. *Method of Defining Syntactic Metalanguage*, 1981.

[12] F.P. Brooks. *The Mythical Man-Month.* Addison-Wesley, Reading, MA, 1975.

[13] Ruven E. Brooks. A theoretical analysis of the role of documentation in the comprehension of computer programs. In *Proc. of the Human Factors in Computer Systems*, pages 125–129. ACM Press, March 1982.

[14] Peter J. Brown. Interactive documentation. *Software — Practice and Experience*, 16(3):291–299, March 1986.

[15] Peter J. Brown. Turning ideas into products: The guide system. In *Proceedings of the Hypertext '87 Workshop*, pages 33–40. The University of North Carolina, November 1987.

[16] Peter J. Brown. Converting help systems to hypertext. *Software — Practice and Experience*, 18(2):163–165, February 1988.

[17] J.K. Buckle. *Software Configuration Management.* MACMILLAN EDUCATIONS LTD., 1982.

[18] Vannevar Bush. As we may think. *Atlantic Monthly*, 176(1):101–108, July 1945. Reprinted in Computer Bulletin, March 1988.

[19] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. In *Proceedings of the Hypertext '87 Workshop*, pages 21–32, November 1987.

[20] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. *Communications of the ACM*, 31(7):856–861, July 1988.

[21] Ned Chapin. Software maintenance: A different view. In *AFIPS 54th National Computer Conference Proceedings*, pages 507–513, 1985.

[22] Y. Chen and C.V. Ramamoorthy. The c information abstractor. In *COMP86*, pages 291–298, October 1986.

[23] Jeff Conklin. Hypertext: An introduction and survey. *IEEE COMPUTER*, pages 17–41, September 1987.

[24] Jeff Conklin. A survey of hypertext. Technical Report STP–356–86, Revision 2, MCC Software Technology Program, December 1987.

[25] Norman Delisle and Mayer Schwartz. NEPTUNE: a hypertext system for cad applications. In *Proceedings of ACM SIGMOD '86*, pages 132–143, 1986.

[26] Norman Delisle and Mayer Schwartz. Contexts — a partitioning concept for hypertext. *ACM Transactions on Office Information Systems*, 5(2):168–186, April 1987.

[27] DOD. DoD automated data systems documentation standards. Technical Report DOD 7935.1-S, Department of Defence, September 1977.

[28] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[29] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[30] FCSC. The software improvement process—its phases and tasks. Technical Report OSD/FCSC-83/006, Office of Software Development and Information Technology, 1983.

[31] Nigel T. Fletton and Malcolm Munro. Redocumenting software systems using hypertext technology. In *Proceedings of the Conference on Software Maintenance — 1988*, pages 54–59, October 1988.

[32] John R. Foster. Software maintenance — an overview. R11 Divisional Memorandum R11/86/013, British Telecom Research Laboratories, August 1986.

[33] John R. Foster and Malcolm Munro. A documentation method based on cross–referencing. In *Proceedings of the Conference on Software Maintenance — 1987*, pages 181–185, 1987.

[34] Pankaj K. Garg and Walt Scacchi. A hypertext system to manage software life cycle documents. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 337–346, 1988.

[35] Adele Goldberg. The influence of an object-oriented language on the programming environment. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 141–171. McGraw–Hill Book Company, 1984.

[36] Danny Goodman. *The Complete HyperCard Handbook.* Bantam Books, New York, September 1987.

[37] John D. Gould. Some psychological evidence on how people debug computer programs. *Internationl Journal of Man–Machine Studies*, 7:151–182, 1975.

[38] Rand P. Hall. Seven ways to cut software maintenance costs. *Datamation*, pages 81,82,84, July 1987.

[39] Jitsuro Harada and Satoshi Sakashita. A documentation tools to visualize program maintainability. In *1983 Software Maintenance Workshop Record*, pages 275–280, 1983.

[40] Ellis Horowitz and Ronald C. Williamson. SODOS: a software documentation support environment—its definition. *IEEE Transactions on Software Engineering*, SE-12(8):849–859, August 1986.

[41] Ellis Horowitz and Ronald C. Williamson. SODOS: a software documentation support environment—its use. *IEEE Transactions on Software Engineering*, SE-12(11):1076–1087, November 1986.

[42] Geoffrey James. *Document Databases.* Van Nostrand Reinhold, New York, 1985.

[43] Gabor Jandrasics. Static analysis of commercial programs with the SOFTDOC system. Technical report, SES Software Engineering Service, Pappelstr. 6, D-8014 Munich/Neubiberg, West Germany, 1981.

[44] Ted Kaehler and Dave Patterson. *A Taste of Smalltalk.* W. W. Norton & Company, 1986.

[45] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, May 1986.

[46] James Martin and Carma McClure. *Software Maintenance: The problem and Its Solutions.* Prentice-Hall, 1983.

[47] Stuart McGowan. Fortune — an ipse documentation tool. Technical report, CAP (UK) Limited, 1987. Alvey Project ALV/PRJ/SE/050.

[48] Douglas Mullin. Software engineer's task analysis. Technical Report 2017/twp/116, CAP (UK) Limited, February 1988. Alvey Project ALV/PRJ/SE/050.

[49] Douglas Mullin and Stuart McGowan. Fortune's functional definition. Technical Report 2017/twp/128, CAP (UK) Limited, January 1988. Alvey Project ALV/PRJ/SE/050.

[50] OWL Ltd. *Guide User's Manual*, 1987.

[51] David L. Parnas. Information distribution aspects of design methodology. In *1971 Proceedings of IFIP Congress*, 1971.

[52] Al Patterson. Understanding and documenting software. In *1983 Software Maintenance Workshop Record*, pages 142–144, 1983. Summary of Session 7.

[53] R.W. Scheifler and J. Gettys. The X window system. *Computer Graphics*, pages 79–109, April 1986.

[54] Norman F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, March 1987.

[55] R. S. Scowen. An introduction and handbook for the standard syntactic metalanguage. NPL Report DITC 19/83, National Physical Laboratory, Teddington, Middlesex TW11 OWL, UK, February 1983.

[56] Margaret E. Singleton. *Automating Code and Documentation Management.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

[57] Harry M. Sneed. Software renewal: A case study. *IEEE Software*, pages 56–63, July 1984.

[58] Ian Sommerville, R. Welland, I. Bennett, and R. Thomson. SOFTLIB—a documentation management system. *Software — Practice and Experience*, 16(2):131–143, February 1986.

[59] Sun Microsystems, Inc. *SunView Programmer's Guide*, September 1986.

[60] Sun Microsystems, Inc. *SunView System Programmer's Guide*, September 1986.

[61] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497. IEEE/ACM, October 1976.

[62] Janet H. Walker. Supporting document development with concordia. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 355–364, 1988.

[63] J.H. Walker. Document examiner: Delivery interface for hypertext documents. In *Proceedings of the Hypertext '87 Workshop*, pages 307–324, 1987.

[64] Nicole Yankelovich, Bernard J. Haan, Norman K. Meyrowitz, and Steven M. Drucker. Intermedia: The concept and construction of a seamless information environment. *IEEE COMPUTER*, 21(1):81–96, January 1988.

[65] Nicole Yankelovich, Norman Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE COMPUTER*, 18(10):15–30, October 1985.

[66] Nicholas Zvegintzov. Nanotrends. *Datamation*, pages 106–116, August 1983.